MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox

Anthony Frank D'Augustine

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Adrian Sandu, Chair
Yang Cao
Lizette Zietsman

December 8, 2017
Blacksburg, Virginia

Keywords: ODE Solver, Tangent Linear Model, Adjoint Model, Sensitivity Analysis, Software

MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox

Anthony Frank D'Augustine

Abstract

Sensitivity analysis quantifies the effect that of perturbations of the model inputs have on the model's outputs. Some of the key insights gained using sensitivity analysis are to understand the robustness of the model with respect to perturbations, and to select the most important parameters for the model. MATLODE is a tool for sensitivity analysis of models described by ordinary differential equations (ODEs). MATLODE implements two distinct approaches for sensitivity analysis: direct (via the tangent linear model) and adjoint. Within each approach, four families of numerical methods are implemented, namely explicit Runge-Kutta, implicit Runge-Kutta, Rosenbrock, and single diagonally implicit Runge-Kutta. Each approach and family has its own strengths and weaknesses when applied to real world problems. MATLODE has a multitude of options that allows users to find the best approach for a wide range of initial value problems. In spite of the great importance of sensitivity analysis for models governed by differential equations, until this work there was no MATLAB ordinary differential equation sensitivity analysis toolbox publicly available. The two most popular sensitivity analysis packages, CVODES [8] and FATODE [10], are geared toward the high performance modeling space; however, no native MATLAB toolbox was available. MATLODE fills this need and offers sensitivity analysis capabilities in MATLAB, one of the most popular programming languages within scientific communities such as chemistry, biology, ecology, and oceanography. We expect that MATLODE will prove to be a useful tool for these communities to help facilitate their research and fill the gap between theory and practice.

MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox

Anthony Frank D'Augustine

General Audience Abstract

Sensitivity analysis is the study of how small changes in a model?s input effect the model's output. Sensitivity analysis provides tools to quantify the impact that small, discrete changes in input values have on the output. The objective of this research is to develop a MATLAB sensitivity analysis toolbox called MATLODE. This research is critical to a wide range of communities who need to optimize system behavior or predict outcomes based on a variety of initial conditions. For example, an analyst could build a model that reflects the performance of an automobile engine, where each part in the engine has a set of initial characteristics. The analyst can use sensitivity analysis to determine which part effects the engine's overall performance the most (or the least), without physically building the engine and running a series of empirical tests. By employing sensitivity analysis, the analyst saves time and money, and since multiple tests can usually be run through the model in the time needed to run just one empirical test, the analyst is likely to gain deeper insight and design a better product. Prior to MATLODE, employing sensitivity analysis without significant knowledge of computational science was too cumbersome and essentially impractical for many of the communities who could benefit from its use. MATLODE bridges the gap between computational science and a variety of communities faced with understanding how small changes in a system's input values effect the systems output; and by bridging that gap, MATLODE enables more large scale research initiatives than ever before.

Acknowledgments

I would like to thank everyone in the Computational Science Laboratory for this incredible journey. I would especially like to thank Dr. Adrian Sandu and Dr. Hong Zhang for giving continuous insight along every step of the way. I would also like to thank my mother, father and sister, without them this would not of been possible.

This work is supported by the National Science Foundation and the Air Force Office of Scientific Research through the awards NSF DMS-1419003, NSF CCF-1613905, NSF ACI-1709727, NSF CCF-0916493 and AFOSR DDDAS 15RT1037.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scientific Context and Problem Definition

Numerical software has a fundamental impact on modern engineering and science. The purpose of engineering and science is to answer a question. To do so rigorously, the scientific method must be applied. The scientific method has five essential steps: make an observation, ask a question, formulate a hypothesis, make a predication and test the prediction. Then repeat, repeat and repeat. In practice, repeating theses steps are both time and economically expensive. To speed up the process and lower the overall costs, numerical software is applied to simulate possible outcomes. For example, in the case of the Apollo 13 incident, the question was how could we get the crew back to earth during a major onboard catastrophe. Famously, Richard Arenstorf developed the Arenstorf Orbit equations describing the spacecraft's path around the moon and earth by harnessing numerical software. Without numerical software, the effort and costs would have been too cumbersome and the Apollo 13 abort probably would have been unsuccessful.

Sensitivity analysis [11] quantifies the impact of input parameters onto the model's outcome. Sensitivity analysis is a key ingredient to understanding complex modeling problems, to performing uncertainty analysis, to optimizing the systems of interest, and to constructing efficient controllers. Consequently, computational tools for sensitivity analysis are much needed across all the science and engineering fields.

However, sensitivity analysis tools are currently only available to a subset of the broader science and engineering community. Currently CVODES [8] and FATODE [10] are the only publicly available sensitivity analysis packages. CVODES is written the programming language C, while FATODE is in FORTRAN. Both are excellent packages, however, their use requires a specific knowledge base in computer science, that not all potential users master.

`MATLAB` is a widely used programing framework that abstracts away much of the lower level

complexity of numerical software. For this reason it has become a de-facto standard in science and engineering computations. Yet, there does not exist a sensitivity analysis package within the MATLAB community. This is sorely needed, as the potential benefits are immense. MATLODE fills this gap by combining the state-of-the-art research within forward integration and sensitivity analysis into the first cohesive MATLAB package. MATLODE takes the conscientious decision to develop the integration and sensitivity toolbox in the native MATLAB programming language.

In practice, it is critical for an application to be implemented in an object-oriented approach for performance, testability, and maintenance reasons. This is why CVODES and FATODE are great choices in large scale computing intensive scenarios, but they lacking lack the object-oriented characteristic. In order to improve research productivity, groups in both private and academic settings typically prototype their concepts in MATLAB, and once the algorithms are satisfactory, they are implemented in a high performance computing environment. Before MATLODE, researchers in need of sensitivity analyses had to rely on packages designed for large scale applications, and therefore invest time in implementation aspects, rather than in conceptualizing..

1.2 Accomplishments of this Work

There are five major accomplishments in this thesis. The first accomplishment is introducing the first MATLAB sensitivity analysis suite. The second accomplishment is introducing the first MATLAB ODE solver suite based on the Runge-Kutta family of integrators. The thesis provides extensive examples on the use of each Runge-Kutta family and sensitivity analysis approach. The third accomplishment is providing a modular design for MATLODE, which is of paramount importance for maintaining a stable code base. The fourth accomplishment is implementing a rigorous testing methodology based on validating MATLODE results against FATODE via the Java Native Interface (JNI). The fifth, and most important, accomplishment is publishing the code in open source format for the benefit of scientists from any field that requires sensitivity analysis.

1.3 Outline of the Thesis

This thesis is composed of four chapters. Chapter 1 discusses the general theory of forward integration solvers and sensitivity analysis. Chapter 2 describes the MATLAB software package MATLODE. Chapter 3 focuses on use cases via two examples based on practical problems. Chapter 4 draws the conclusions of this work. The Appendix contains the software User's Manual.

In the theory chapter, the forward integration section describes the explicit Runge-Kutta,

implicit Runge-Kutta, singly diagonally implicit Runge-Kutta and Rosenbrock solvers in their general form. The following section introduces what sensitivity analysis means in the context of initial value problems.

The bulk of the thesis is focused on the `MATLODE` software development. In the software chapter a comparison between `MATLAB`'s forward integration solvers is made from a semantics and performance perspective. The semantics concentrates on the similarities and usabilities considerations during the design process. Within the performance analysis, since `MATLAB` does not offer sensitivity analysis in their forward integration solvers, we cannot compare sensitivity analysis capabilities. Instead an analysis of only the forward integration solvers is discussed.

After the comparison, two use cases is conducted to give a bigger picture of what `MATLODE`'s capabilities. The Shallow Water Equations are used to demonstrate a large scale simulation using `MATLODE`'s forward integration solvers. Then an interesting application that uses `MATLODE`'s sensitivity analysis capabilities – parameter estimation – is solved.

Chapter 2

Theoretical Background

Initial-value ordinary differential equation problems are described as follows:

$$y\prime = f(t, y, p), \quad y(t_0) = y_0(p). \tag{2.1}$$

where $t \in \mathbb{R}$ is time and $y(t) \in \mathbb{R}$ represents the model state and $p \in \mathbb{R}^m$ is the time-independent model parameters. Let the function depending on the final state vector be defined as:

$$\Psi = g(y(t_N), p) \tag{2.2}$$

.

We now define the sensitivity analysis matrix as

$$S = \frac{d\Psi}{dp} \tag{2.3}$$

where S is the derivative of the solution with respect to parameters.

2.1 Forward Integration Methods for the Underlying ODE System

The general s-stage Runge-Kutta method takes the form

$$y_{n+1} = y_n + h \sum_{j=1}^{s} b_j f(T_j, Y_j)$$

where

$$T_i = t_n + c_j h$$

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} f(T_j, Y_j) \tag{2.4}$$

and the coefficients are

$$A = [a_{ij}]_{1 \leq i,j \leq s}, \quad b = [b_i]_{1 \leq i \leq s}, \quad c = [c_i]_{1 \leq i \leq s} \tag{2.5}$$

for $i = 1, \ldots, s$ and $j = 1, \ldots, s$. The variables t and h represent time and step size respectively. The function f describes the model. The characteristics of function f determines which family of forward integration is best suited for the model.

2.1.1 Fully Implicit Runge-Kutta Methods

A common alternative form for implicit Runge-Kutta reads

$$y_{n+1} = y_n + h \sum_{i=1} b_i k_i$$

where

$$k_i = f(T_i, y_n + h \sum_{j=1}^{s} a_{ij} k_j)$$

and the implicit Runge-Kutta butcher tableau is

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \ldots & a_{1s} \\ c_2 & a_{21} & a_{22} & \ldots & a_{2s} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \ddots & a_{ss} \\ \hline & b_1 & b_2 & \ldots & b_s \end{array}$$

2.1.2 Explicit Runge-Kutta Methods

Applying the general s-stage Runge-Kutta method 2.4 and the explicit Runge-Kutta coefficients, the coefficients read

$$A = [a_{ij}]_{1 \leq j < i \leq s}, \quad b = [b_i]_{1 \leq i \leq s}, \quad c = [c_i]_{1 < i \leq s} \tag{2.6}$$

for $i = 1, \ldots, s$ and $j = 1, \ldots, s-1$ which is equivalent to the explicit Runge-Kutta butcher tableau

$$\begin{array}{c|cccc} 0 & 0 & \ldots & \ldots & 0 \\ c_2 & a_{21} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_s & a_{s1} & \ldots & a_{s-1,s} & 0 \\ \hline & b_1 & b_2 & \ldots & b_s \end{array}$$

2.1.3 Singly Diagonally Implicit Runge-Kutta Methods

Again applying the general s-stage Runge-Kutta method 2.4 and the singly diagonally implicit Runge-Kutta coefficients, the coefficients read

$$A = [a_{ij}]_{1 \leq j \leq i \leq s}, \quad b = [b_i]_{1 \leq i \leq s}, \quad c = [c_i]_{1 \leq i \leq s} \tag{2.7}$$

where

$$A = [a_{ij}]_{j=i \leq s} = \gamma \tag{2.8}$$

for $i = 1, \ldots, s$ and $j = 1, \ldots, s-1$ which is equivalent to the explicit Runge-Kutta butcher tableau

$$\begin{array}{c|cccc} c_1 & \gamma & \ldots & \ldots & 0 \\ c_2 & a_{21} & \gamma & \ldots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_s & a_{s1} & \ldots & a_{s-1,s} & \gamma \\ \hline & b_1 & \ldots & b_s & \end{array}$$

2.1.4 Rosenbrock Methods

The general s-stage Rosenbrock method takes the form

$$y_{n+1} = y_n + \sum_{j=1}^{s} b_j k_j \tag{2.9}$$

where

$$k_i = hf(y_0 + \sum_{j=1}^{i-1} \alpha_{ij} k_j) + hJ \sum_{j=1}^{i} \gamma_{ij} k_j \tag{2.10}$$

for the parameters $i = 1, \ldots, s$. α,γ and β are determined apriori based on the method order. In practice we use the form [6]

$$y_{n+1} = y_n + \sum_{j=1}^{s} m_j k_j \tag{2.11}$$

where

$$\left[\frac{1}{h\gamma_{ii}} I - J(t_n, y_n)\right] k_i = f(t_n + \alpha_i h, y_n + \sum_{j=1}^{i-1} a_{ij} k_j) + \sum_{j=1}^{i-1} (\frac{c_{ij}}{h}) k_j + h\gamma_i f_t(t_n, y_n) \tag{2.12}$$

$$\alpha_{ij} = (\alpha_{ij})\Gamma^{-1} \tag{2.13}$$

$$(m_1, \ldots, m_s) = (\beta_1, \ldots, \beta_s)\Gamma^{-1} \tag{2.14}$$

$$\Gamma = \gamma_{ij} \tag{2.15}$$

for $i = 1, \ldots, s$. Once again, the coefficients are determined by the desired method order.

2.2 Sensitivity Analysis

Generalizing 2.1 via the discrete forward model we obtain

$$y_{n+1} = \Phi^n(y_n, p) \tag{2.16}$$

for $n = 1, \ldots, N-1$.

2.2.1 Tangent Linear Model

Let the discrete tangent linear model be defined by the following

$$\dot{y}_{n+1} = \Psi_y^n(y_n, p)\dot{y}_n + \Psi_p^n(y_n, p) \tag{2.17}$$

for $n = 0, \ldots, N-1$. Computing each column of the discrete tangent linear model we obtain the ℓ column of the sensitivity matrix

$$\frac{\partial \Psi}{\partial p_\ell} = g_y(y_n, p)(\dot{y}_n)_\ell + g_{p_\ell}(y_n, p) \tag{2.18}$$

for $\ell = 1, \ldots, m$.

Fully Implicit Runge-Kutta Methods

For the fully implicit Runge-Kutta tangent linear integrator, `MATLODE` offers two options for solving the linear system. Option one is to solve the linear system directly using `MATLAB` backslash linear solver. Option two, is to solve the linear system using Newton iterations. From experience, Newton iterations is sufficient for solving the linear solving and hence by default Newton iterations is chosen.

Singly Diagonally Implicit Runge-Kutta Methods

Both the forward integration and the tangent linear model require solving a linear system. To increase performance, LU decomposition is applied by default. During the forward integration phase, the LU decomposition is stored for the tangent linear model phase, hence increasing performance since no additional computation is required. If the LU decomposition option does not suffice, `MATLODE` also offers to solve the linear system via Newton iteration as an alternative.

2.2.2 Adjoint Model

Let the discrete adjoint model be defined by the following

$$(\lambda_i)_N = (g_i)_y^T(y_N)$$

$$(\mu_i)_N = (g_i)_p^T(y_N) \tag{2.19}$$

at the final time step. Then iterating backwards in time, solve for the previous λ and μ as described below

$$(\lambda_i)_n = \Phi_y^n(y_n, p)^T(\lambda_i)_{n+1} \tag{2.20}$$

$$(\mu_i)_n = \Phi_p^n(y_n, p)^T(\lambda_i)_{n+1} \tag{2.21}$$

for $n = N-1, \ldots, 0$. Computing the sensitivity for each ith state variable to all parameters we obtain the adjoint matrix components

$$\frac{d\Psi_i}{dp} = (\mu_i)_0^T + (\lambda_i)_0^T \frac{dy_0}{dp}. \tag{2.22}$$

Aggregating all adjoint matrix components we obtain the full sensitivity matrix $d\Phi/dp$.

For all adjoint model implementations, we reuse the forward code base. This in turn allows all of the functionality introduced in the forward integrators to be applied to the adjoint integrators.

Chapter 3

The MATLODE Software Environment

In this section, a top-level overview of MATLODE's basic functionality is given. MATLODE offers forward integration and sensitivity analysis. For each forward and sensitivity analysis integrator, explicit Runge-Kutta, implicit Runge-Kutta, Rosenbrock and singly diagonally implicit Runge-Kutta families of integrators are available. In addition, MATLODE offers both tangent linear and adjoint sensitivity analysis. Lastly, MATLODE also allows for custom Runge-Kutta coefficients to be added to the suite for particularly arcane initial value problems.

From an engineering perspective, MATLODE features enhanced logging capabilities. For example, printing time steps and user input option struct warnings. MATLODE has gone to great lengths to ensure valid user inputs into the integrators. MATLODE also features a lightweight installation process. All installation entails is downloading the toolbox and adding the root directory to the active path. The most important feature of MATLODE is its extendibility for future development given its architecture. In the next section, we further describe how all the pieces fit together to form MATLODE.

3.1 MATLODE Architecture

All MATLODE integrators are broken down into three abstraction layers. The first is called the option layer. The second is characterized as the messenger layer and the third, the core algorithm layer. In this section, we describe in greater detail the role of each layer.

3.1.1 Option Layer

The option layer is the control center for all MATLODE integrators. Through the use of key-value pairs, the user can choose whether or not to fine tune the solver to their model. The option layer is composed of three phases. The first phase displays a warning to the

ODE Solver	Characteristic	Forward	Sensitivty	Method
MATLODE_ERK_ADJ_Integrator	Nonstiff	Yes	Adjoint	Runge-Kutta
MATLODE_RK_ADJ_Integrator	Stiff	Yes	Adjoint	Runge-Kutta
MATLODE_ROS_ADJ_Integrator	Stiff	Yes	Adjoint	Rosenbrock
MATLODE_SDIRK_ADJ_Integrator	Stiff	Yes	Adjoint	Runge-Kutta
MATLODE_ERK_FWD_Integrator	Nonstiff	Yes	-	Runge-Kutta
MATLODE_RK_FWD_Integrator	Stiff	Yes	-	Runge-Kutta
MATLODE_ROS_FWD_Integrator	Stiff	Yes	-	Rosenbrock
MATLODE_SDIRK_FWD_Integrator	Stiff	Yes	-	Runge-Kutta
MATLODE_ERK_TLM_Integrator	Nonstiff	No	Tangent Linear	Runge-Kutta
MATLODE_RK_TLM_Integrator	Stiff	No	Tangent Linear	Runge-Kutta
MATLODE_ROS_TLM_Integrator	Stiff	No	Tangent Linear	Rosenbrock
MATLODE_SDIRK_TLM_Integrator	Stiff	No	Tangent Linear	Runge-Kutta

Table 3.1: MATLODE Integrators.

Value	Coefficient	Stages	Order	Stability Properties
0 (Default)	Dropri5	7	5	conditionally-stable
1	Erk23	3	2	conditionally-stable
2	Erk3 Heun	4	3	conditionally-stable
3	Erk43	5	4	conditionally-stable
4	Dropri5	7	5	conditionally-stable
5	Dropri853	12	8	conditionally-stable

Table 3.2: MATLODE ERK Coefficients.

MATLAB console if the option parameter is not used for the desired integrator. The second phase, configures the options struct to the integrator's predefined settings. In general, the predefined settings are considered sufficient for the majority of models. Where finer control is necessary, the third phase overwrites the predefined parameters with user defined values. Hence, giving the user ultimate control over the solver.

See tables 3.9, 3.3, 3.4, 3.5, 3.6, 3.7 and 3.8 for available option parameter key-value pairs.

3.1.2 Messenger Layer

The messenger layer is composed of two tasks. Initiate the option and core algorithm layers. Both the option and messenger layer are exposed to the user. The messenger layer is what the user actually calls to integrate a model, hiding the complexities in the core algorithm layer.

In addition to these two tasks, the messenger layer has the responsibility of calling both the forward and adjoint in sensitivity analysis. From a software design perspective, the mes-

Value	Coefficient	Stages	Order	Stability Properties
0 (Default)	Lobatto3C	3	4	L-stable
1	Radua2A	3	5	L-stable
2	Lobatto3C	3	4	L-stable
3	Gauss	3	6	weakly L-stable
4	Radau1A	3	5	L-stable

Table 3.3: MATLODE RK Coefficients.

Value	Coefficient	Stages	Order	Stability Properties
0 (Default)	Ros4	4	4	L-stable
1	Ros2	2	2	L-stable
2	Ros3	3	3	L-stable
3	Ros4	4	4	L-stable
4	Rodas3	4	3	L-stable
5	Rodas4	5	4	L-stable

Table 3.4: MATLODE ROS Coefficients.

Value	Coefficient	Stages	Order	Stability Properties
0 (Default)	Sdirk4A	5	4	L-stable
1	Sdirk2A	2	2	L-stable
2	Sdirk2B	2	2	L-stable
3	Sdirk3A	3	2	L-stable
4	Sdirk4A	5	4	L-stable
5	Sdirk4B	5	4	L-stable

Table 3.5: MATLODE SDIRK Coefficients.

senger layer allows for all features implemented in the forward integrators to be available in the Adjoint sensitivity analysis integrators as well by code reuse. This approach is extremely beneficial for code maintainability since each component, forward and Adjoint, can be isolated and tested.

3.1.3 Core Algorithm Layer

The core algorithm layer is defined in two phases, integration and error control. The integration uses the option struct defined in the option layer to make decisions as the model is propagated in time. Within the integration phase, the model is projected by the current time step approximating the model's state. After the integration phase, the next step size is determined in the error control phase. The combination of the integration and error phases is what propagates the model in time.

3.2 Using MATLODE

In this section, a basic example of MATLODE in action and how to use the User's manual is described. The reader is encouraged to follow along to better understand how to use MATLODE

3.2.1 Example

For the following example, Arenstorf Orbit is used as a toy problem to illustrate MATLODE_ERK_FWD_Integrat functionalities and features. To initially setup Arenstorf Orbit, exexute the MATLAB commands below to load the input parameters into the workspace.

Key	Value	Description	Family
AbsTol	double, double[]	Absolute error tolerance for forward integrators	{ERK, RK, ROS, SDIRK}
AbsTol_ADJ	double, double[]	Absolute Newton iteration tolerance for solving adjoint system	{}
AbsTol_TLM	double double[]	Absolute error estimation for TLM at Newton stages	{}
ChunkSize	integer	Appended memory block size	{ERK, RK, ROS, SDIRK}
DirectADJ	boolean	Determines whether direct adjoint sensitivity analysis is performed	{}
DirectTLM	boolean	Determines whether direct tangent linear sensitivity analysis is performed	{}
DisplayStats	boolean	Determines whether statistics are displayed	{ERK, RK, ROS, SDIRK}
DisplaySteps	boolean	Determines whether steps are displayed	{ERK, RK, ROS, SDIRK}
DRDP	function	Derivative of r w.r.t. parameters	{}
DRDY	function	Derivative of r w.r.t. y vector	{}
FaxMax	double	Step size upper bound change ratio	{ERK, RK, ROS, SDIRK}
FacMin	double	step size lower bound change ratio	{ERK, RK, ROS, SDIRK}
FacSafe	double	Factor by which the new step is slightly smaller than the predicted value	{ERK, RK, ROS, SDIRK}
FDAprox	boolean	Determines whether Jacobian vector products by finite difference is used	{}
Gustafsson	boolean	An alternative error controller approach which may be advantageous on the model characteristics	{}
Hess_vec	function	Hv	{}
Hesstr_vec	function	$H^T v$	{}
Hesstr_vec_f_py	function	xxx	{}
Hesstr_vec_r	function	xxx	{}
Hesstr_vec_r_py	function	xxx	{}
Hmax	double	Step size upper bound	{ERK, RK, ROS, SDIRK}
Hmin	double	Step size lower bound	{ERK, RK, ROS, SDIRK}
Hstart	double	Initial step size	{ERK, RK, ROS, SDIRK}
Jacobian	function	User defined function: Jacobian	{RK, ROS, SDIRK}
Jacp	function	User defined function: $\frac{df}{dp}$	{}
Lambda	double[]	Adjoint sensitivity matrix	{}
Max_no_steps	integer	Maximum number of steps upper bound	{ERK, RK, ROS, SDIRK}
Method	integer	Determines which coefficient to use	{ERK, RK, ROS, SDIRK}
Mu	double[]	Mu vector for sensitivity analysis	{}
NewtonMaxit	integer	Maximum number of Newton iterations	{RK, SDIRK}
NewtonTol	double	Newton method stopping criterion	{RK, SDIRK}
NP	integer	Number of parameters	{}
QFun	function	r function	{}
QMax	double	Predicted step size to current step size upper bound ratio	{ERK, RK, ROS, SDIRK}
QMin	double	Predicted step size to current step size lower bound ratio	{ERK, RK, ROS, SDIRK}
Quadrature	double[]	Initial quadrature	{}
RelTol	double, double[]	Relative tolerance	{ERK, RK, ROS, SDIRK}
RelTol_ADJ	double, double[]	Relative Newton iteration tolerance for solving adjoint system	{}
RelTol_TLM	double, double[]	Relative error estimation for TLM at Newton stages	{}
SaveLU	boolean	Determines whether to save during LU factorization	{}
SdirkError	boolean	Alternative error criterion	{}
StartNewton	boolean	Determines whether Newton iterations are performed	{RK, SDIRK}
StoreCheckpoint	boolean	Determines whether intermediate values are stored	{ERK, RK, ROS, SDIRK}
ThetaMin	double	Factor deciding whether the Jacobian should be recomputed	{}
TLMNewtonEst	boolean	Determines whether to user a tangent linear scaling factor in Newton iteration	{}
TLMtruncErr	boolean	Determines whether to incorporate sensitivity truncation error	{}
WarningConfig	boolean	Determines whether warning messages are displayed during option configuration	{ERK, RK, ROS, SDIRK}
Y_TLM	double[]	Contains the sensitivities of Y with respect to the specified coefficients	{}

Table 3.6: `MATLODE` FWD options.

Key	Value	Description	Family
AbsTol	double, double[]	Absolute error tolerance for forward integrators	{ERK, RK, ROS, SDIRK}
AbsTol_ADJ	double, double[]	Absolute Newton iteration tolerance for solving adjoint system	{}
AbsTol_TLM	double double[]	Absolute error estimation for TLM at Newton stages	{ROS}
ChunkSize	integer	Appended memory block size	{ERK, RK, ROS, SDIRK}
DirectADJ	boolean	Determines whether direct adjoint sensitivity analysis is performed	{}
DirectTLM	boolean	Determines whether direct tangent linear sensitivity analysis is performed	{RK, ROS, SDIRK}
DisplayStats	boolean	Determines whether statistics are displayed	{ERK, RK, ROS, SDIRK}
DisplaySteps	boolean	Determines whether steps are displayed	{ERK, RK, ROS, SDIRK}
DRDP	function	Derivative of r w.r.t. parameters	{}
DRDY	function	Derivative of r w.r.t. y vector	{}
FaxMax	double	Step size upper bound change ratio	{ERK, RK, ROS, SDIRK}
FacMin	double	step size lower bound change ratio	{ERK, RK, ROS, SDIRK}
FacSafe	double	Factor by which the new step is slightly smaller than the predicted value	{ERK, RK, ROS, SDIRK}
FDAprox	boolean	Determines whether Jacobian vector products by finite difference is used	{ERK, RK, SDIRK}
Gustafsson	boolean	An alternative error controller approach which may be advantageous on the model characteristics	{RK}
Hess_vec	function	Hv	{}
Hesstr_vec	function	$H^T v$	{}
Hesstr_vec_f_py	function	xxx	{}
Hesstr_vec_r	function	xxx	{}
Hesstr_vec_r_py	function	xxx	{}
Hmax	double	Step size upper bound	{ERK, RK, ROS, SDIRK}
Hmin	double	Step size lower bound	{ERK, RK, ROS, SDIRK}
Hstart	double	Initial step size	{ERK, RK, ROS, SDIRK}
Jacobian	function	User defined function: Jacobian	{RK, ROS, SDIRK}
Jacp	function	User defined function: $\frac{df}{dp}$	{ERK}
Lambda	double[]	Adjoint sensitivity matrix	{}
Max_no_steps	integer	Maximum number of steps upper bound	{ERK, RK, ROS, SDIRK}
Method	integer	Determines which coefficient to use	{ERK, RK, ROS, SDIRK}
Mu	double[]	Mu vector for sensitivity analysis	{}
NewtonMaxit	integer	Maximum number of Newton iterations	{RK, SDIRK}
NewtonTol	double	Newton method stopping criterion	{RK, SDIRK}
NP	integer	Number of parameters	{}
QFun	function	r function	{}
QMax	double	Predicted step size to current step size upper bound ratio	{ERK, RK, ROS, SDIRK}
QMin	double	Predicted step size to current step size lower bound ratio	{ERK, RK, ROS, SDIRK}
Quadrature	double[]	Initial quadrature	{}
RelTol	double, double[]	Relative tolerance	{ERK, RK, ROS, SDIRK}
RelTol_ADJ	double, double[]	Relative Newton iteration tolerance for solving adjoint system	{}
RelTol_TLM	double, double[]	Relative error estimation for TLM at Newton stages	{ROS}
SaveLU	boolean	Determines whether to save during LU factorization	{}
SdirkError	boolean	Alternative error criterion	{RK}
StartNewton	boolean	Determines whether Newton iterations are performed	{RK, SDIRK}
StoreCheckpoint	boolean	Determines whether intermediate values are stored	{ERK, RK, ROS, SDIRK}
ThetaMin	double	Factor deciding whether the Jacobian should be recomputed	{RK, SDIRK}
TLMNewtonEst	boolean	Determines whether to user a tangent linear scaling factor in Newton iteration	{RK, SDIRK}
TLMtruncErr	boolean	Determines whether to incorporate sensitivity truncation error	{ERK, RK, ROS, SDIRK}
WarningConfig	boolean	Determines whether warning messages are displayed during option configuration	{ERK, RK, ROS, SDIRK}
Y_TLM	double[]	Contains the sensitivities of Y with respect to the specified coefficients	{ERK, RK, ROS, SDIRK}

Table 3.7: `MATLODE` TLM options.

Key	Value	Description	Family
AbsTol	double, double[]	Absolute error tolerance for forward integrators	{ERK, RK, ROS, SDIRK}
AbsTol_ADJ	double, double[]	Absolute Newton iteration tolerance for solving adjoint system	{SDIRK}
AbsTol_TLM	double double[]	Absolute error estimation for TLM at Newton stages	{}
ChunkSize	integer	Appended memory block size	{ERK, RK, ROS, SDIRK}
DirectADJ	boolean	Determines whether direct adjoint sensitivity analysis is performed	{RK, SDIRK}
DirectTLM	boolean	Determines whether direct tangent linear sensitivity analysis is performed	{}
DisplayStats	boolean	Determines whether statistics are displayed	{ERK, RK, ROS, SDIRK}
DisplaySteps	boolean	Determines whether steps are displayed	{ERK, RK, ROS, SDIRK}
DRDP	function	Derivative of r w.r.t. parameters	{ERK, RK, ROS, SDIRK}
DRDY	function	Derivative of r w.r.t. y vector	{ERK, RK, ROS, SDIRK}
FaxMax	double	Step size upper bound change ratio	{ERK, RK, ROS, SDIRK}
FacMin	double	step size lower bound change ratio	{ERK, RK, ROS, SDIRK}
FacSafe	double	Factor by which the new step is slightly smaller than the predicted value	{ERK, RK, ROS, SDIRK}
FDAprox	boolean	Determines whether Jacobian vector products by finite difference is used	{}
Gustafsson	boolean	An alternative error controller approach which may be advantageous on the model characteristics	{RK}
Hess_vec	function	Hv	{ERK, RK, ROS, SDIRK}
Hesstr_vec	function	$H^T v$	{ERK, RK, ROS, SDIRK}
Hesstr_vec_f_py	function	xxx	{ERK, RK, ROS, SDIRK}
Hesstr_vec_r	function	xxx	{ERK, RK, ROS, SDIRK}
Hesstr_vec_r_py	function	xxx	{ERK, RK, ROS, SDIRK}
Hmax	double	Step size upper bound	{ERK, RK, ROS, SDIRK}
Hmin	double	Step size lower bound	{ERK, RK, ROS, SDIRK}
Hstart	double	Initial step size	{ERK, RK, ROS, SDIRK}
Jacobian	function	User defined function: Jacobian	{ERK, RK, ROS, SDIRK}
Jacp	function	User defined function: $\frac{df}{dp}$	{ERK, RK, ROS, SDIRK}
Lambda	double[]	Adjoint sensitivity matrix	{ERK, RK, ROS, SDIRK}
Max_no_steps	integer	Maximum number of steps upper bound	{ERK, RK, ROS, SDIRK}
Method	integer	Determines which coefficient to use	{ERK, RK, ROS, SDIRK}
Mu	double[]	Mu vector for sensitivity analysis	{}
NewtonMaxit	integer	Maximum number of Newton iterations	{RK, SDIRK}
NewtonTol	double	Newton method stopping criterion	{RK, SDIRK}
NP	integer	Number of parameters	{}
QFun	function	r function	{ERK, RK, ROS, SDIRK}
QMax	double	Predicted step size to current step size upper bound ratio	{ERK, RK, ROS, SDIRK}
QMin	double	Predicted step size to current step size lower bound ratio	{ERK, RK, ROS, SDIRK}
Quadrature	double[]	Initial quadrature	{ERK, RK, ROS, SDIRK}
RelTol	double, double[]	Relative tolerance	{ERK, RK, ROS, SDIRK}
RelTol_ADJ	double, double[]	Relative Newton iteration tolerance for solving adjoint system	{SDIRK}
RelTol_TLM	double, double[]	Relative error estimation for TLM at Newton stages	{}
SaveLU	boolean	Determines whether to save during LU factorization	{SDIRK}
SdirkError	boolean	Alternative error criterion	{RK}
StartNewton	boolean	Determines whether Newton iterations are performed	{RK, SDIRK}
StoreCheckpoint	boolean	Determines whether intermediate values are stored	{ERK, RK, ROS, SDIRK}
ThetaMin	double	Factor deciding whether the Jacobian should be recomputed	{}
TLMNewtonEst	boolean	Determines whether to user a tangent linear scaling factor in Newton iteration	{}
TLMtruncErr	boolean	Determines whether to incorporate sensitivity truncation error	{}
WarningConfig	boolean	Determines whether warning messages are displayed during option configuration	{ERK, RK, ROS, SDIRK}
Y_TLM	double[]	Contains the sensitivities of Y with respect to the specified coefficients	{}

Table 3.8: `MATLODE` ADJ options.

```
Ode_Function = @arenstorfOrbit_Function;

Time_Interval = [ 0 17.0652166];

Y0 = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that the model is loaded in the workspace, one performs a forward explicit Runge-Kutta integration using the prebuilt default settings.

```
[  , Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0)];
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');

Y(end,:));
```

The following will be printed to the console.

```
solution at Time_Interval(2)

0.9894, -0.0081 -1.1139 -1.3474
```

To save the model state at each time step, one needs to initialize a `MATLODE` option struct to store the fine tuning settings. The `(key, value)` pair associated for saving the model state at each time step is denoted as `('storeCheckpoint', true)` or `('storeCheckpoint', false)` depending on whether or not one wants to explicitly fine tune the integrator. In this case, the intermediary time step values are stored executing the command below.

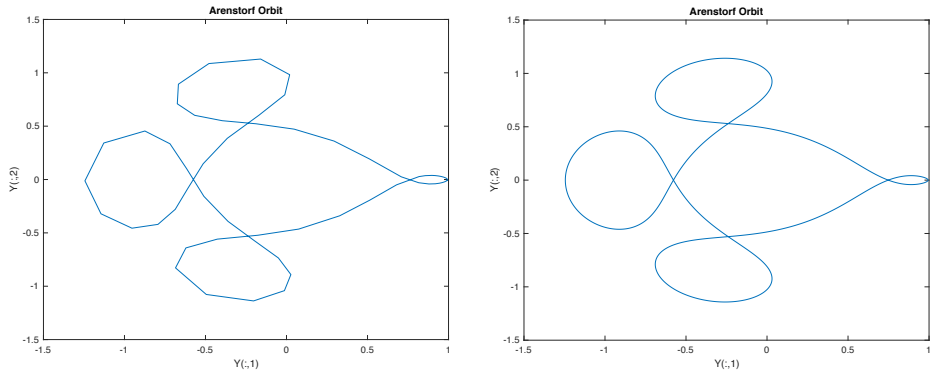
```
Options = MATLODE_OPTIONS('storeCheckpoint', true);
```

To run `MATLODE_ERK_FWD_Integrator` using the fine tuning, one needs to insert the option struct into the integrator's fourth parameter position.

```
[  , Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options);
```

After plotting the results, one can now visualize the model.

```
figure(1);
```



(a) Arenstorf Orbit (rough) (b) Arenstorf Orbit (smooth)

Figure 3.1: Results after running example problem.

```
plot(Y(:,1), Y(:,2));

title('Arenstorf Orbit');

xlabel('Y(:,1)');

ylabel('Y(:,2)');
```

To obtain a smoother graphical repusentation, one can further tighten the error tolerances. To tighten the relative and absolute error tolerances, one fine tunes the option struct. Since the option struct is already in the workspace, one adds the relative and absolute `(key, value)` pair to the option struct. Then plot the results.

```
Options = MATLODE_OPTIONS(Options, 'AbsTol' 1e-12, 'RelTol', 1e-12);

[ T, Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options);

figure(2);

plot(Y(:,1), Y(:,2));

title('Arenstorf Orbit');

xlabel('Y(:,1)');

ylabel('Y(:,2)');
```

For more examples, see the attached appendix.

Section	Description
Contents	The table of contents for given section.
Syntax	The method signatures.
Input Parameters	The available input parameters and argument positions.
Output Parameters	The available output parameters and positions.
Description	Describes what the integrator is.
Example	A brief simplistic example of how to execute the integrator.
Contact Information	People who developed the integrator. If the user has questions, please use one of these email addresses.
Reference	If conducting research, this section describes how to reference the integrator.
Major Modification History	A history of which developer modified the integrator.

Table 3.9: MATLODE user manual sections.

3.2.2 User Manual

The user manual should be viewed as a jumpstart guide. The purpose of the user manual is to describe how each integrator can be utilized. Table 3.9 describes the user manual's section. The user is encouraged to look through the appendix prior to using MATLODE.

3.3 Comparison with MATLAB's native ODE suite

3.3.1 Interface

MATLAB's naming convention is described as

$$[\texttt{t},\texttt{y}] = \texttt{ode}[\texttt{orderInfo}][\texttt{additionalInfo}](\texttt{odefun},\texttt{tspan},\texttt{y0},\texttt{options})$$

where orderInfo and addiontalInfor corresponds to the chosen method's theoretical properties and

$$\texttt{option} = \texttt{odeset}(\texttt{'name1'},\texttt{value1},\texttt{'name2'},\texttt{value2},...)$$

where ('name1',value1), ('name2',value2),... are key-value pairs. For example, breaking ode23s into three parts, ode, 23 and s correspond to the suite the user is using, the orders of the internal methods and MATLAB's classification of the integration, in this case stiff.

MATLODE takes the a slightly different approach, explicitly including the family and implementation in the function's name.MATLODE's naming convention is described as

$$[\texttt{t},\texttt{y}] = \texttt{MATLODE\_}[\texttt{family}]\texttt{\_}[\texttt{implementation}]\texttt{\_Integrator}(\texttt{odefun},\texttt{tspan},\texttt{y0},\texttt{options})$$

where family $\in$ {'ERK','RK','ROS',SDIRK' } and implementation $\in$ {'FWD','TLM','ADJ'}. By including the family name, the user is able to assess which family is most advantageous given the model's characteristics and parameters.MATLODE's option naming convention is

$$\texttt{option} = \texttt{MATLODE\_OPTIONS}(\texttt{'name1'},\texttt{value1},\texttt{'name2'},\texttt{value2},...)$$

where ('name1',value1), ('name2',value2),... are a key-value pairs.

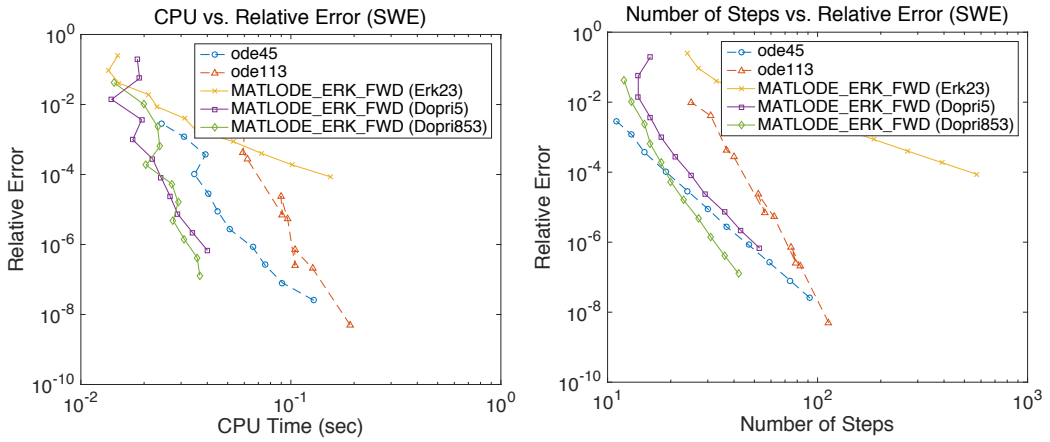


Figure 3.2: Above is a comparison between `MATLAB` and `MATLODE` non-stiff integrators. `ode23` is also a non-stiff integrator in the `MATLAB` ordinary differential equation suite, but is not graphed because the CPU time is significantly large for shallow water equations. For reproducibility, run `ACM_TOMS_Publication_F3.m` in the `MATLODE` Toolbox.

3.3.2 Performance

Comparing integration software is quite difficult and can often be considered an art at times. Every model has its own particular characteristics that determine the integrators performance. For this comparison we have chosen a non-stiff and stiff model.

For the non-stiff model, the shallow water equations using `MATLODE` and `MATLAB` integrators were configured to their default configurations varying the family of integration. From figure 3.2, we see that `MATLODE` outperforms `MATLAB`'s general purpose solvers for the shallow water equations. `MATLODE`'s most efficient solver for the shallow water equations is Dopri5 and Dopri857.

CBM4 was used as the stiff model. CBM4 is a large scale realistic model with extremely stiff components and consists of 32 species and 82 reactions.`MATLODE`'s best performing solver is Rosenbrock with Rodas4 coefficients while `MATLAB`'s best integrator is ode15s. For CBM4 the results are comparable.

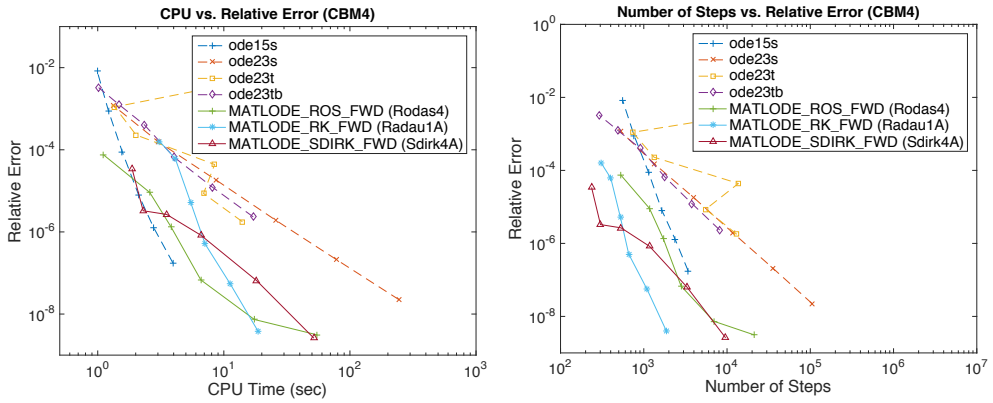


Figure 3.3: Above is a comparison between `MATLAB` and `MATLODE` stiff integrators. The reference solution was calculated using ode15s for all integrators except for its self. Instead, ode23s was used to calculate the error associated with ode15s.

Chapter 4

Use Cases

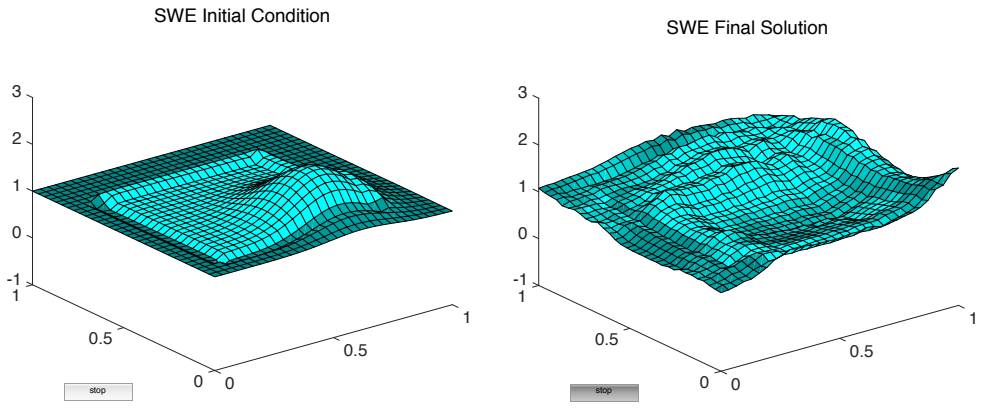
In practice, MATLODE has many applications and is only bounded by what ordinary differential equations cannot describe. The two use cases demonstrated below are the Shallow Water Equations and Carbon Bond Mechanism version IV (CBM4). For Shallow Water Equations, we demonstrate how MATLODE's forward integration modules are harnessed to further understand the phenomenon. Using CBM4, we also show how MATLODE's sensitivity analysis capabilities can be used for parameter estimation.

4.1 Forward Integration: Shallow Water Equations

As with all initial value problems, the initial conditions are required to kickstart the solver. As scene in figure 4.1, the initial condition for the Shallow Water Equation we are analyzing is a slight swell. Given prior knowledge the non-stiff nature of the Shallow Water Equations we propagate the model using explicit Runge-Kutta. MATLODE saves each time step in two matrices. The first matrix is an array where the value at each index tells the user when each model state was captured. The second matrix archives the model state at each time step. The index of the time matrix corresponds to the row of the model state matrix. Therefore, to play an animation one iterates over the rows of the model state matrix. As we can see from figure 4.1, the final solution is significantly rougher than the initial condition. This is due to error propagation. The reader is encouraged to tighten the tolerance levels in MATLODE to obtain a smoother smoother at the final time.

4.2 Parameter Estimation: CBM4

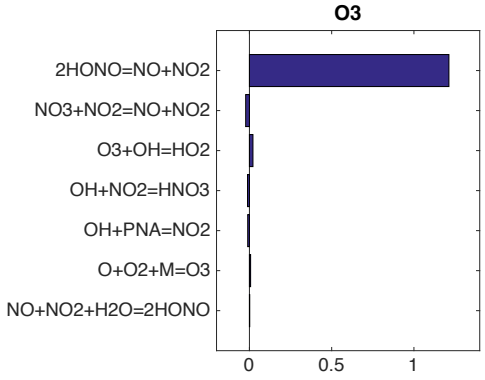
We now consider application of parameter estimation using MATLODE's sensitivity analysis capabilities. The experiment uses the singly diagonally implicit Runge-Kutta for forward



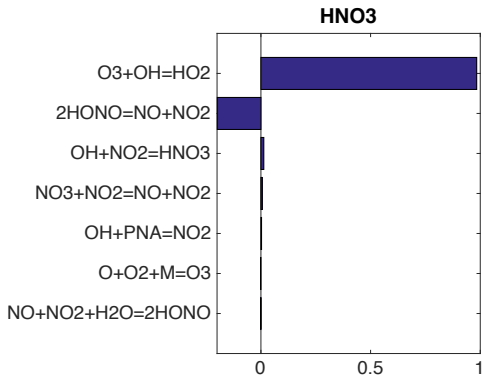
(a) Initial condition (b) Final solution (after 3.2s)

Figure 4.1: Solutions of the shallow water equations model. For reproducibility, run `ACM_TOMS_Publication_F2.m` in the MATLODE Toolbox.

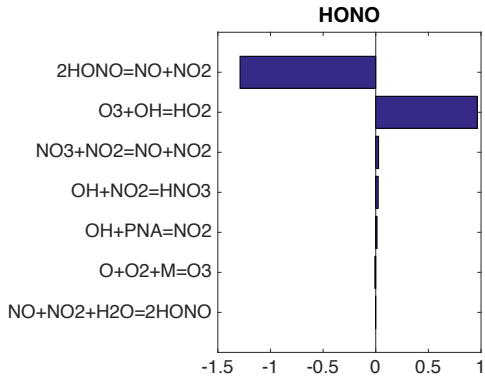
integration and the adjoint sensitivities with respect to the chemical reaction coefficients. In this experiment, the most important model variables are O_3, NO_2, $HONO$, N_2O_5 and HNO_3. To understand the most influential reaction rates for this model variables, we sort the sensitivities for each model variable of interest by magnitude. The most influential reactions are depicted in 4.2.



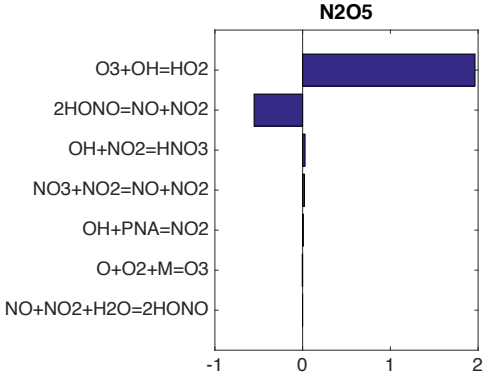
(a) Sensitivities of O_3.



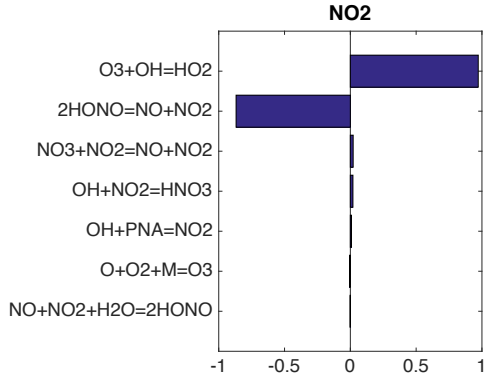
(b) Sensitivities of HNO_3.



(c) Sensitivities of $HONO$.



(d) Sensitivities of N_2O_5.



(e) Sensitivities of NO_2.

Figure 4.2: Adjoint sensitivities of several chemical species with respect to reaction rates. Each panel illustrates the top seven parameters that affect the corresponding chemical species the most. The reactions are marked on the left axes. The less influential reactions are not shown for brevity.

Chapter 5

Conclusions and Future Work

MATLODE fills a critical gap in the engineering and science communities by providing an extensive native MATLAB sensitivity analysis implementation. Application that benefit include, but are not limited to, optimizing models and parameter estimation. The MATLODE framework allows for future extensions of the current functionality by harnessing key software engineering principles, e.g., using GitLab in the workflow. This will make biannual bug fixes and feature enhancements possible for years to come. MATLODE is available online at http://www.matlode.com, and is expected that the scientific community will use and benefit from the technology.

Future work will include the implementation of event detection capabilities in the software. Computational modelers often do not know the exact time when an event occurs. The modeler instead knows the model state or event of interest. For example, consider the well-known predator-prey model. Suppose we want to halt the forward integration not at a final time but rather when the prey population reaches a certain number. This is rather simple for a single purpose solver but can get cumbersome for a general package. The concept Inversion of Control (IoC) is the key to implementing this feature. In the future, MATLODE developers will explore this design pattern to allow users to have greater control within their development environment. More importantly, this will offer large scale applications the greater ability to integrate with MATLODE at production quality code level.

In the future, we also plan to explore applications of MATLODE to real-time systems such as algorithmic trading strategies. As one can imagine, the application of parameter estimation in an algorithmic trading strategy environment can give a trader a significant advantage when determining whether the model continues to obey the initial assumptions. The key challenge will be to calculate the sensitivities fast enough, and provide them prior to the actual event occurence.

Chapter 6

Bibliography

[1] Choose A Solver. (2015). Available at `http://www.mathworks.com/help/simulink/ug/choosing-a-solver.html`.

[2] Ordinary Differential Equations. (2015). Available at `http://www.mathworks.com/help/matlab/math/ordinary-differential-equations.html`.

[3] User Stories By Industry. (2015). Available at `http://www.mathworks.com/company/userstories/industry.html`.

[4] Blom, D. S., et al. "Rosenbrock time integration for unsteady flow simulations." Coupled Problems 2013: Proceedings of the 5th International Conference on Computational Methods for Coupled Problems in Science and Engineering, Ibiza, Spain, 17-19 June 2013. CIMNE, 2013.

[5] Ernst Hairer, Syvert P Norsett, and Gerhard Wanner (2009). Solving ordinary differential equations I: Nonstiff problems (Springer Series In Computational Mathematics).

[6] Ernst Hairer and Gerhard Wanner (2005). Solving ordinary differential equations II. Stiff and differential- algebraic problems (Springer Series in Computational Mathematics).

[7] Radu Serban (2005). "`SUNDIALSTB`, a Matlab Interface to `SUNDIALS`." Technical Report. Lawrence Livermore National Lab., Livermore, CA (US).

[8] Radu Serban and Alan C. Hindmarsh (2005). "`CVODES`: the sensitivity-enabled ODE solver in `SUNDIALS`." In ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. American Society of Mechanical Engineers, pp. 257–269.

[9] Lawrence F. Shampine and Mark W. Reichelt (1997). "The MATLAB ODE Suite." SIAM Journal on Scientific Computing, Vol. 18, No. 1, pp. 1-22. DOI:http://dx.doi.org/10.1137/S1064827594276424.

[10] Hong Zhang and Adrian Sandu (2014). "FATODE: A library for forward, adjoint and tangent linear Integration of stiff systems." SIAM Journal on Scientific Computing, Vol. 36, No. 5 ,pp. C504-C523.

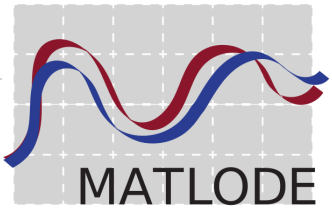
[11] Andrea Saltelli, Karen Chan, and E. Marian Scott, eds (2000). "Sensitivity analysis." Vol. 1, Wiley, New York, pp 1-9.

Appendix A

MATLODE's User Guide

User Guide

Up: MATLODE Toolbox



Contents

ODE Solver	Example	Problem Characteristics	Forward Solution	Tangent Linear Sensitivity	Adjoint Sensitivity	Method
MATLODE_ERK_ADJ_Integrator	MATLODE_Example_ERK_ADJ_Integrator	Nonstiff differential equations	✓	✗	✓	Runge-Kutta
MATLODE_RK_ADJ_Integrator	MATLODE_Example_RK_ADJ_Integrator	Stiff differential equations	✓	✗	✓	Runge-Kutta
MATLODE_ROS_ADJ_Integrator	MATLODE_Example_ROS_ADJ_Integrator	Stiff differential equations	✓	✗	✓	Rosenbrock
MATLODE_SDIRK_ADJ_Integrator	MATLODE_Example_SDIRK_ADJ_Integrator	Stiff differential equations	✓	✗	✓	Runge-Kutta
MATLODE_ERK_FWD_Integrator	MATLODE_Example_ERK_FWD_Integrator	Nonstiff differential equations	✓	✗	✗	Runge-Kutta
MATLODE_RK_FWD_Integrator	MATLODE_Example_RK_FWD_Integrator	Stiff differential equations	✓	✗	✗	Runge-Kutta
MATLODE_ROS_FWD_Integrator	MATLODE_Example_ROS_FWD_Integrator	Stiff differential equations	✓	✗	✗	Rosenbrock
MATLODE_SDIRK_FWD_Integrator	MATLODE_Example_SDIRK_FWD_Integrator	Stiff differential equations	✓	✗	✗	Runge-Kutta
MATLODE_ERK_TLM_Integrator	MATLODE_Example_ERK_TLM_Integrator	Nonstiff differential equations	✓	✓	✗	Runge-Kutta
MATLODE_RK_TLM_Integrator	MATLODE_Example_RK_TLM_Integrator	Stiff differential equations	✓	✓	✗	Runge-Kutta
MATLODE_ROS_TLM_Integrator	MATLODE_Example_ROS_TLM_Integrator	Stiff differential equations	✓	✓	✗	Rosenbrock
MATLODE_SDIRK_TLM_Integrator	MATLODE_Example_SDIRK_TLM_Integrator	Stiff differential equations	✓	✓	✗	Runge-Kutta

Methods

Method: Explicit Runge-Kutta (ERK)

Value	Configuration	Stages	Order	Stability Properties
0 (default)	Dopri5	7	5	conditionally-stable
1	Erk23	3	2	conditionally-stable
2	Erk3 Heun	4	3	conditionally-stable
3	Erk43	5	4	conditionally-stable
4	Dopri5	7	5	conditionally-stable
5	Dopri853	12	8	conditionally-stable

Method: Implicit Runge-Kutta (RK)

Value	Configuration	Stages	Order	Stability Properties
0 (default)	Lobatto3C	3	4	L-stable
1	Radua2A	3	5	L-stable
2	Lobatto3C	3	4	L-stable
3	Gauss	3	6	weakly L-stable
4	Radau1A	3	5	L-stable

Method: Rosenbrock (ROS)

Value	Configuration	Stages	Order	Stability Properties
0 (default)	Ros4	4	4	L-stable
1	Ros2	2	2	L-stable
2	Ros3	3	3	L-stable
3	Ros4	4	4	L-stable
4	Rodas3	4	3	L-stable
5	Rodas4	6	4	L-stable

Method: Singly Diagonally Implicit Runge-Kutta (SDIRK)

Value	Configuration	Stages	Order	Stability Properties
0 (default)	Sdirk4A	5	4	L-stable
1	Sdrik2A	2	2	L-stable
2	Sdirk2B	2	2	L-stable
3	Sdirk3A	3	2	L-stable
4	Sdirk4A	5	4	L-stable
5	Sdirk4B	5	4	L-stable

Return to Top

Forward Integrators

MATLODE_OPTIONS: Forward Integrator

Key	Value	Description	MATLODE_ERK_FWD_Integrator	MATLODE_RK_FWD_Integrator	MATLODE_ROS_FWD_Integrator	MATLODE_SDIRK_FWD_Integrator
AbsTol	double, double[]	Absolute error tolerance for Forward integrators	✓	✓	✓	✓
AbsTol_ADJ	double, double[]	Absolute Newton iteration tolerance for solving adjoint system	✗	✗	✗	✗
AbsTol_TLM	double, double[]	Absolute error estimation for TLM at Newton stages	✗	✗	✗	✗
ChunkSize	integer	Appended memory block size	✓	✓	✓	✓
DirectADJ	boolean	Determines whether direct adjoint sensitivity analysis is performed	✗	✗	✗	✗
DirectTLM	boolean	Determines whether direct tangent linear sensitivity analysis is performed	✗	✗	✗	✗
DisplayStats	boolean	Determines whether statistics are displayed	✓	✓	✓	✓
DisplaySteps	boolean	Determines whether steps are displayedd	✓	✓	✓	✓
DRDP	function handle	Derivitive of r w.r.t. parameters	✗	✗	✗	✗
DRDY	function handle	Derivitive of r w.r.t. y vector	✗	✗	✗	✗
FacMax	double	Step size upper bound change ratio	✓	✓	✓	✓
FacMin	double	Step size lower bound change ratio	✓	✓	✓	✓
FacSafe	double	Factor by which the new step is slightly smaller than the predicted value	✓	✓	✓	✓
FDAprox	boolean	Determines whether Jacobian vector products by finite difference is used	✗	✗	✗	✗
Gustafsson	boolean	An alternative	✗	✗	✗	✗

		error controller approach which may be advantageous depending on the model characteristics				
Hess_vec	function handle	H * v	✗	✗	✗	✗
Hesstr_vec	function handle	H^T * v	✗	✗	✗	✗
Hesstr_vec_f_py	function handle	(d(f_p^T * u)/dy) * k	✗	✗	✗	✗
Hesstr_vec_r	function handle	(d(r_y^T * u)/dy) * k	✗	✗	✗	✗
Hesstr_vec_r_py	function handle	(d(r_p^T *u)/dy) *k	✗	✗	✗	✗
Hmax	double	Step size upper bound	✓	✓	✓	✓
Hmin	double	Step size lower bound	✓	✓	✓	✓
Hstart	double	Initial step size	✓	✓	✓	✓
ITOL	boolean	Depreciated: Tolerances are scalor or vector	✓	✓	✓	✓
Jacobian	function handle	User defined function: Jacobian	✗	✓	✓	✓
Jacp	function handle	User defined function: df/dp	✗	✗	✗	✗
Lambda	double[]	Adjoint sensitivity matrix	✗	✗	✗	✗
MatrixFree	boolean	Determines whether Jacobian is approximated	✗	✗	✗	✓
Max_no_steps	integer	Maximum number of steps upper bound	✓	✓	✓	✓
Method	integer	Determines which coefficients to use	✓	✓	✓	✓
Mu	double[]	Mu vector for sensitivity analysis	✗	✗	✗	✗
NBasisVectors	integer	Number of basis vectors	✗	✗	✗	✗
NewtonMaxIt	integer	Maximum number of newton iterations performed	✗	✓	✗	✓
NewtonTol	double	Newton method stopping criterion	✗	✓	✗	✓
NP	integer	Number of parameters	✗	✗	✗	✗
QFun	function handle	r function	✗	✗	✗	✗
Qmax	double	Predicted step size to	✓	✓	✓	✓

		current step size upper bound ratio				
Qmin	double	Predicted step size to current step size lower bound ratio	✓	✓	✓	✓
Quadrature	double[]	Initial Quadrature	✗	✗	✗	✗
RelTol	double, double[]	Relative error tolerance	✓	✓	✓	✓
RelTol_ADJ	double, double[]	Relative Newton iteration tolerance for solving adjoint system	✗	✗	✗	✗
RelTol_TLM	double, double[]	Relative error estimation for TLM at Newton stages	✗	✗	✗	✗
SaveLU	boolean	Determines whether to save during LU factorization	✗	✗	✗	✗
SdirkError	boolean	Alternative error criterionn	✗	✗	✗	✗
StartNewton	boolean	Determines whether newton iterations are performed	✗	✓	✗	✓
StoreCheckpoint	boolean	Determines whether intermediate values are stored	✓	✓	✓	✓
ThetaMin	double	Factor deciding whether the Jacobian should be recomputed	✗	✗	✗	✗
TLMNewtonEst	boolean	Determines whether to user a tangent linear scaling factor in newton interation	✗	✗	✗	✗
TLMtruncErr	boolean	Determiens whether to incorpate sensitivity truncation error	✗	✗	✗	✗
WarningConfig	boolean	Determines whether warning messages are displayed during option configuration	✓	✓	✓	✓
Y_TLM	double[]	Contains the sensitivities of Y with respect to the specified coefficients	✗	✗	✗	✗

Return to Top

Tangent Linear Integrators

MATLODE_OPTIONS: Tangent Linear Integrator

Key	Value	Description	MATLODE_ERK_TLM_Integrator	MATLODE_RK_TLM_Integrator	MATLODE_ROS_TLM_Integrator	MATLODE_SDIRK_TLM_Integrator
AbsTol	double, double[]	Absolute error tolerance for Forward integrators	✓	✓	✓	✓
AbsTol_ADJ	double, double[]	Absolute Newton iteration tolerance for solving adjoint system	✗	✗	✗	✗
AbsTol_TLM	double, double[]	Absolute error estimation for TLM at Newton stages	✗	✗	✓	✗
ChunkSize	integer	Appended memory block size	✓	✓	✓	✓
DirectADJ	boolean	Determines whether direct adjoint sensitivity analysis is performed	✗	✗	✗	✗
DirectTLM	boolean	Determines whether direct tangent linear sensitivity analysis is performed	✗	✓	✓	✓
DisplayStats	boolean	Determines whether statistics are displayed	✓	✓	✓	✓
DisplaySteps	boolean	Determines whether steps are displayedd	✓	✓	✓	✓
DRDP	function handle	Derivitive of r w.r.t. parameters	✗	✗	✗	✗
DRDY	function handle	Derivitive of r w.r.t. y vector	✗	✗	✗	✗
FacMax	double	Step size upper bound change ratio	✓	✓	✓	✓
FacMin	double	Step size lower bound change ratio	✓	✓	✓	✓
FacSafe	double	Factor by which the new step is slightly smaller than the predicted value	✓	✓	✓	✓
FDAprox	boolean	Determines whether Jacobian vector products by finite difference is used	✓	✓	✗	✗
Gustafsson	boolean	An alternative error controller approach which may be advantageous depending on	✗	✓	✗	✗

		the model characteristics				
Hess_vec	function handle	H * v	✗	✗	✗	✗
Hesstr_vec	function handle	H^T * v	✗	✗	✗	✗
Hesstr_vec_f_py	function handle	(d(f_p^T * u)/dy) * k	✗	✗	✗	✗
Hesstr_vec_r	function handle	(d(r_y^T * u)/dy) * k	✗	✗	✗	✗
Hesstr_vec_r_py	function handle	(d(r_p^T *u)/dy) *k	✗	✗	✗	✗
Hmax	double	Step size upper bound	✓	✓	✓	✓
Hmin	double	Step size lower bound	✓	✓	✓	✓
Hstart	double	Initial step size	✓	✓	✓	✓
ITOL	boolean	Depreciated: Tolerances are scalor or vector	✓	✓	✓	✓
Jacobian	function handle	User defined function: Jacobian	✓	✓	✓	✓
Jacp	function handle	User defined function: df/dp	✗	✗	✗	✗
Lambda	double[]	Adjoint sensitivity matrix	✗	✗	✗	✗
MatrixFree	boolean	Determines whether Jacobian is approximated	✗	✗	✗	✓
Max_no_steps	integer	Maximum number of steps upper bound	✓	✓	✓	✓
Method	integer	Determines which coefficients to use	✓	✓	✓	✓
Mu	double[]	Mu vector for sensitivity analysis	✗	✗	✗	✗
NBasisVectors	integer	Number of basis vectors	✗	✗	✗	✗
NewtonMaxIt	integer	Maximum number of newton iterations performed	✗	✓	✗	✓
NewtonTol	double	Newton method stopping criterion	✗	✓	✗	✓
NP	integer	Number of parameters	✗	✗	✗	✗
QFun	function handle	r function	✗	✗	✗	✗
Qmax	double	Predicted step size to current step size upper bound ratio	✓	✓	✓	✓
Qmin	double	Predicted step size to	✓	✓	✓	✓

		current step size lower bound ratio				
Quadrature	double[]	Initial Quadrature	✗	✗	✗	✗
RelTol	double, double[]	Relative error tolerance	✓	✓	✓	✓
RelTol_ADJ	double, double[]	Relative Newton iteration tolerance for solving adjoint system	✗	✗	✗	✗
RelTol_TLM	double, double[]	Relative error estimation for TLM at Newton stages	✗	✗	✓	✗
SaveLU	boolean	Determines whether to save during LU factorization	✗	✗	✗	✗
SdirkError	boolean	Alternative error criterionn	✗	✓	✗	✗
StartNewton	boolean	Determines whether newton iterations are performed	✗	✓	✗	✓
StoreCheckpoint	boolean	Determines whether intermediate values are stored	✓	✓	✓	✓
ThetaMin	double	Factor deciding whether the Jacobian should be recomputed	✗	✓	✗	✓
TLMNewtonEst	boolean	Determines whether to user a tangent linear scaling factor in newton interation	✗	✓	✗	✓
TLMtruncErr	boolean	Determiens whether to incorpate sensitivity truncation error	✓	✓	✓	✓
WarningConfig	boolean	Determines whether warning messages are displayed during option configuration	✓	✓	✓	✓
Y_TLM	double[]	Contains the sensitivities of Y with respect to the specified coefficients	✓	✓	✓	✓

Return to Top

Adjoint Integrators

MATLODE_OPTIONS: Adjoint Integrator

Key	Value	Description	MATLODE_ERK_ADJ_Integrator	MATLODE_RK_ADJ_Integrator	MATLODE_ROS_ADJ_Integrator	MATLODE_SDIRK_ADJ_Integrator

AbsTol	double, double[]	Absolute error tolerance for Forward integrators	✓	✓	✓	✓
AbsTol_ADJ	double, double[]	Absolute Newton iteration tolerance for solving adjoint system	✗	✗	✗	✓
AbsTol_TLM	double, double[]	Absolute error estimation for TLM at Newton stages	✗	✗	✗	✗
ChunkSize	integer	Appended memory block size	✓	✓	✓	✓
DirectADJ	boolean	Determines whether direct adjoint sensitivity analysis is performed	✗	✓	✗	✓
DirectTLM	boolean	Determines whether direct tangent linear sensitivity analysis is performed	✗	✗	✗	✗
DisplayStats	boolean	Determines whether statistics are displayed	✓	✓	✓	✓
DisplaySteps	boolean	Determines whether steps are displayedd	✓	✓	✓	✓
DRDP	function handle	Derivitive of r w.r.t. parameters	✓	✓	✓	✓
DRDY	function handle	Derivitive of r w.r.t. y vector	✓	✓	✓	✓
FacMax	double	Step size upper bound change ratio	✓	✓	✓	✓
FacMin	double	Step size lower bound change ratio	✓	✓	✓	✓
FacSafe	double	Factor by which the new step is slightly smaller than the predicted value	✓	✓	✓	✓
FDAprox	boolean	Determines whether Jacobian vector products by finite difference is used	✗	✗	✗	✗
Gustafsson	boolean	An alternative error controller approach which may be advantageous depending on the model characteristics	✗	✓	✗	✗
Hess_vec	function handle	H * v	✓	✓	✓	✓

Hesstr_vec	function handle	H^T * v	✓	✓	✓	✓
Hesstr_vec_f_py	function handle	(d(f_p^T * u)/dy) * k	✓	✓	✓	✓
Hesstr_vec_r	function handle	(d(r_y^T * u)/dy) * k	✓	✓	✓	✓
Hesstr_vec_r_py	function handle	(d(r_p^T *u)/dy) *k	✓	✓	✓	✓
Hmax	double	Step size upper bound	✓	✓	✓	✓
Hmin	double	Step size lower bound	✓	✓	✓	✓
Hstart	double	Initial step size	✓	✓	✓	✓
ITOL	boolean	Depreciated: Tolerances are scalor or vector	✓	✓	✓	✓
Jacobian	function handle	User defined function: Jacobian	✓	✓	✓	✓
Jacp	function handle	User defined function: df/dp	✓	✓	✓	✓
Lambda	double[]	Adjoint sensitivity matrix	✓	✓	✓	✓
MatrixFree	boolean	Determines whether Jacobian is approximated	✗	✗	✗	✓
Max_no_steps	integer	Maximum number of steps upper bound	✓	✓	✓	✓
Method	integer	Determines which coefficients to use	✓	✓	✓	✓
Mu	double[]	Mu vector for sensitivity analysis	✓	✓	✓	✓
NBasisVectors	integer	Number of basis vectors	✗	✗	✗	✗
NewtonMaxIt	integer	Maximum number of newton iterations performed	✗	✓	✗	✓
NewtonTol	double	Newton method stopping criterion	✗	✓	✗	✓
NP	integer	Number of parameters	✗	✗	✗	✗
QFun	function handle	r function	✓	✓	✓	✓
Qmax	double	Predicted step size to current step size upper bound ratio	✓	✓	✓	✓
Qmin	double	Predicted step size to current step size lower bound ratio	✓	✓	✓	✓
Quadrature	double[]	Initial Quadrature	✓	✓	✓	✓

RelTol	double, double[]	Relative error tolerance	✓	✓	✓	✓
RelTol_ADJ	double, double[]	Relative Newton iteration tolerance for solving adjoint system	✗	✗	✗	✓
RelTol_TLM	double, double[]	Relative error estimation for TLM at Newton stages	✗	✗	✗	✗
SaveLU	boolean	Determines whether to save during LU factorization	✗	✓	✗	✓
SdirkError	boolean	Alternative error criterionn	✗	✓	✗	✗
StartNewton	boolean	Determines whether newton iterations are performed	✗	✓	✗	✓
StoreCheckpoint	boolean	Determines whether intermediate values are stored	✓	✓	✓	✓
ThetaMin	double	Factor deciding whether the Jacobian should be recomputed	✗	✗	✗	✗
TLMNewtonEst	boolean	Determines whether to user a tangent linear scaling factor in newton interation	✗	✗	✗	✗
TLMtruncErr	boolean	Determiens whether to incorpate sensitivity truncation error	✗	✗	✗	✗
WarningConfig	boolean	Determines whether warning messages are displayed during option configuration	✓	✓	✓	✓
Y_TLM	double[]	Contains the sensitivities of Y with respect to the specified coefficients	✗	✗	✗	✗

Return to Top

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Major Modification History

Date	Developer	Email	Action

1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_ERK_ADJ_Integrator

Contents

Syntax

```
                                MATLODE_ERK_ADJ_Integrator
                  [ T, Y, Sens ] = MATLODE_ERK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Mu, Stats ] = MATLODE_ERK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Mu`: Mu term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and adjoint sensitivity using an Explicit Runge-Kutta (ERK) method.

`MATLODE_ERK_ADJ_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_ERK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and adjoint sensitivity.

`[T, Y, Sens, Quad, Mu, Stats]` = MATLODE_ERK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, sensitivity, quadrature, mu and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ERK_ADJ_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_Lambda          = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a adjoint explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Lambda are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda);
[ T, Y, Sens ] = MATLODE_ERK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and adjoint sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_ERK_ADJ_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_RK_ADJ_Integrator

Contents

Syntax

```
                                MATLODE_RK_ADJ_Integrator
                [ T, Y, Sens ] = MATLODE_RK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Mu, Stats ] = MATLODE_RK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Mu`: Mu term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and adjoint sensitivity using an Implicit Runge-Kutta (RK) method.

`MATLODE_RK_ADJ_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_RK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and adjoint sensitivity.

`[T, Y, Sens, Quad, Mu, Stats]` = MATLODE_RK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, sensitivity, quadrature, mu and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_RK_ADJ_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_Lambda          = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a adjoint implicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Lambda are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda);
[ T, Y, Sens ] = MATLODE_RK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and adjoint sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_RK_ADJ_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

[3] E. Hairer and G. Wanner, Solving Ordinary Differential Equations II. Stiff and Differential- Algebraic Problems, Springer Series in Computational Mathematics, Berlin, 1991.

`Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.`

```
Computational Science Laboratory, Virginia Tech.
```


Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE

Published with MATLAB® R2014b

MATLODE_ROS_ADJ_Integrator

Contents

Syntax

```
                                MATLODE_ROS_ADJ_Integrator
                  [ T, Y, Sens ] = MATLODE_ROS_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Mu, Stats ] = MATLODE_ROS_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Mu`: Mu term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and adjoint sensitivity using a Rosenbrock (ROS) method.

`MATLODE_ROS_ADJ_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_ROS_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and adjoint sensitivity.

`[T, Y, Sens, Quad, Mu, Stats]` = MATLODE_ROS_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, sensitivity, quadrature, mu and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ROS_ADJ_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_Lambda          = eye(4);
Ode_HessTr          = @arenstorfOrbit_Hesstr_vec;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a adjoint rosenbrock integration using MATLODE's prebuilt default settings. We note that a Jacobian, Lambda and Hessian transpose are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Hesstr_vec',Ode_HessTr);
[ T, Y, Sens ] = MATLODE_ROS_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and adjoint sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_ROS_ADJ_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE

Published with MATLAB® R2014b

MATLODE_SDIRK_ADJ_Integrator

Contents

Syntax

```
                                  MATLODE_SDIRK_ADJ_Integrator
                  [ T, Y, Sens ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Mu, Stats ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Mu`: Mu term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and adjoint sensitivity using a Rosenbrock (SDIRK) method.

`MATLODE_SDIRK_ADJ_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and adjoint sensitivity.

`[T, Y, Sens, Quad, Mu, Stats]` = MATLODE_SDIRK_ADJ_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, sensitivity, quadrature, mu and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_SDIRK_ADJ_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_Lambda          = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a adjoint singly diagonally implicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian, Lambda and Hessian transpose are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda);
[ T, Y, Sens ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and adjoint sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_SDIRK_ADJ_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE

Published with MATLAB® R2014b

MATLODE_ERK_FWD_Integrator

Contents

Syntax

```
                MATLODE_ERK_FWD_Integrator
       [T, Y] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0)
[T, Y, Stats] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0)
       [T, Y] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)
[T, Y, Stats] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) using a Explicit Runge-Kutta (ERK) method.

`MATLODE_ERK_FWD_Integrator` displays the available methods associated with the explicit Runge Kutta forward integrator.

`[T, Y] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0)` computes the ODE solution at the final time using the default parameters.

`[T, Y, Stats] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0)` computes the ODE solution at the final time using the default parameters and returns computational statistics.

`[T, Y] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration.

`[T, Y, Stats] = MATLODE_ERK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and returns the computation statistics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ERK_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a forward explicit Runge-Kutta integration using MATLODE's prebuilt default settings.

```
[ T, Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function,Time_Interval,Y0);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Forward Integration -> MATLODE_Example_ERK_FWD_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on

Scientific Computing, 36(5), C504-C523, 2014.

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_RK_FWD_Integrator

Contents

Syntax

```
                MATLODE_RK_FWD_Integrator
       [T, Y] = MATLODE_RK_FWD_Integrator(Ode_Function, Time_Interval, Y0)
[T, Y, Stats] = MATLODE_RK_FWD_Integrator(Ode_Function, Time_Interval, Y0)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) using a Implicit Runge-Kutta (RK) method.

`MATLODE_RK_FWD_Integrator` displays the available methods associated with the implicit Runge Kutta forward integrator.

`[T, Y] = MATLODE_RK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration.

`[T, Y, Stats] = MATLODE_RK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and returns the computation statistics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ERK_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a forward implicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian is required for an implicit method.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian);
[ T, Y ] = MATLODE_RK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Forward Integration -> MATLODE_Example_RK_FWD_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_ROS_FWD_Integrator

Contents

Syntax

```
                MATLODE_ROS_FWD_Integrator
       [T, Y] = MATLODE_ROS_FWD_Integrator(Ode_Function, Time_Interval, Y0)
[T, Y, Stats] = MATLODE_ROS_FWD_Integrator(Ode_Function, Time_Interval, Y0)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) using a Rosenbrock (ROS) method.

`MATLODE_ROS_FWD_Integrator` displays the available methods associated with the rosenbrock forward integrator.

`[T, Y] = MATLODE_ROS_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration.

`[T, Y, Stats] = MATLODE_ROS_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and returns the computation statistics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ROS_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a forward exponential integration using MATLODE's prebuilt default settings. We note that a Jacobian is required for an rosenbrock method.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian);
[ T, Y ] = MATLODE_ROS_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Forward Integration -> MATLODE_Example_ROS_FWD_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504-C523, 2014.

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt.edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_SDIRK_FWD_Integrator

Contents

Syntax

```
                MATLODE_SDIRK_FWD_Integrator
       [T, Y] = MATLODE_SDIRK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)
[T, Y, Stats] = MATLODE_SDIRK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) using a Singly Diagonally Implicit Runge Kutta (SDIRK) method.

MATLODE_SDIRK_FWD_Integrator displays the available methods associated with the Singly Diagonally Implicit Runge Kutta forward integrator.

[T, Y] = MATLODE_SDIRK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options) computes the ODE solution with respect to the user supplied options configuration.

[T, Y, Stats] = MATLODE_SDIRK_FWD_Integrator(Ode_Function, Time_Interval, Y0, Options) computes the ODE solution with respect to the user supplied options configuration and returns the computation statistics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_SDIRK_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a forward exponential integration using MATLODE's prebuilt default settings. We note that a Jacobian is required for an singly diagonally implicit runge-kutta method.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian);
[ T, Y ] = MATLODE_SDIRK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Forward Integration -> MATLODE_Example_SDIRK_FWD_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt.edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_ERK_TLM_Integrator

Contents

Syntax

```
                              MATLODE_ERK_TLM_Integrator
                [ T, Y, Sens ] = MATLODE_ERK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Stats ] = MATLODE_ERK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and tangent linear sensitivity using an Explicit Runge-Kutta (ERK) method.

`MATLODE_ERK_TLM_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_ERK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and tangent linear sensitivity.

`[T, Y, Sens, Quad, Stats]` = MATLODE_ERK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, tangent linear sensitivity, quadrature and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ERK_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a tangent linear explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Y_TLM are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM);
[ T, Y, Sens ] = MATLODE_ERK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and tangent linear sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_ERK_TLM_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_RK_TLM_Integrator

Contents

Syntax

```
                                MATLODE_RK_TLM_Integrator
                  [ T, Y, Sens ] = MATLODE_RK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Stats ] = MATLODE_RK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and tangent linear sensitivity using an Implicit Runge-Kutta (RK) method.

`MATLODE_RK_TLM_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_RK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and tangent linear sensitivity.

`[T, Y, Sens, Quad, Stats]` = MATLODE_RK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, tangent linear sensitivity, quadrature and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_RK_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a tangent linear explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Y_TLM are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM);
[ T, Y, Sens ] = MATLODE_RK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and tangent linear sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_RK_TLM_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

`Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.`

```
Computational Science Laboratory, Virginia Tech.
©2015 Virginia Tech Intellectual Properties, Inc.
```

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_ROS_TLM_Integrator

Contents

Syntax

```
                              MATLODE_ROS_TLM_Integrator
                [ T, Y, Sens ] = MATLODE_ROS_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Stats ] = MATLODE_ROS_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and tangent linear sensitivity using an Rosenbrock (ROS) method.

`MATLODE_ROS_TLM_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_ROS_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and tangent linear sensitivity.

`[T, Y, Sens, Quad, Stats]` = MATLODE_ROS_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, tangent linear sensitivity, quadrature and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_RK_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_HessVec         = @arenstorfOrbit_Hess_vec;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a tangent linear explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Y_TLM are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM,'Hess_vec',Ode_HessVec);
[ T, Y, Sens ] = MATLODE_ROS_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and tangent linear sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_ROS_TLM_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

`Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.`

```
Computational Science Laboratory, Virginia Tech.
```

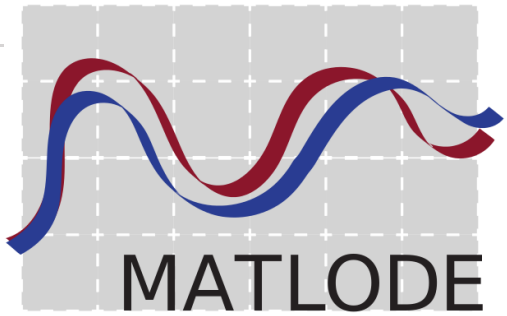

Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_SDIRK_TLM_Integrator

Contents



Syntax

```
                                MATLODE_SDIRK_TLM_Integrator
              [ T, Y, Sens ] = MATLODE_SDIRK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
[ T, Y, Sens, Quad, Stats ] = MATLODE_SDIRK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)
```

Input Parameters

`Ode_Function`: model function

`Time_Interval`: time span

`Y0`: initial model state vector

`Options`: MATLODE option struct

Output Parameters

`T`: saved time snapshots

`Y`: saved model state vectors

`Sens`: Sensitivity matrix

`Quad`: Quadrature term

`Stats`: integrator statistics

Description

Driver file to solve the system y' = F(t,y) and tangent linear sensitivity using an Singly Diagonally Implicit Runge-Kutta (SDIRK) method.

`MATLODE_SDIRK_TLM_Integrator` displays the available methods associated with the exponential forward integrator.

`[T, Y, Sens] = MATLODE_SDIRK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)` computes the ODE solution with respect to the user supplied options configuration and tangent linear sensitivity.

`[T, Y, Sens, Quad, Stats]` = MATLODE_SDIRK_TLM_Integrator(Ode_Function, Time_Interval, Y0, Options)| computes the ODE solution with respect to the user supplied options configuration, tangent linear sensitivity, quadrature and statisitics.

Example

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_SDIRK_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Now that we have our model loaded in our workspace, we can perform a tangent linear explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Y_TLM are required for sensitivity analysis.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM);
[ T, Y, Sens ] = MATLODE_SDIRK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution and tangent linear sensitivty at Time_Interval(2)');
disp(Y);
disp(Sens);
```

For addition examples, see Help -> Supplemental Software -> Examples -> Sensitivity Analysis -> MATLODE_Example_SDIRK_TLM_Integrator.

Contact Information

Dr. Adrian Sandu | Phone: (540) 231-2193 | Email: sandu@cs.vt.edu

Tony D'Augustine | Phone: (540) 231-6186 | Email: adaug13@vt.edu

Computational Science Laboratory | Phone: (540) 231-6186

Reference

[1] Tony D'Augustine, Adrian Sandu. MATLODE: A MATLAB ODE Solver and Sensitivity Analysis Toolbox. Submitted to ACM TOMS.

[2] Hong Zhang, Adrian Sandu. FATODE: a library for forward, adjoint and tangent linear integration of ODEs, SIAM Journal on Scientific Computing, 36(5), C504–C523, 2014

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

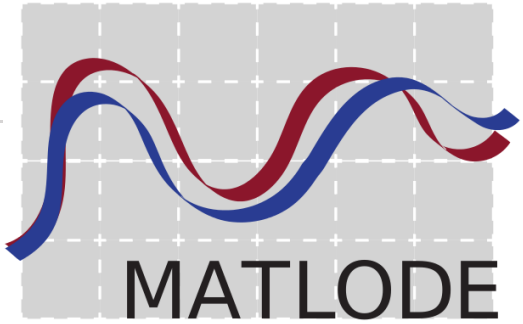
Major Modification History

Date	Developer	Email	Action
1/1/2014	Tony D'Augustine	adaug13@vt,edu	Release MATLODE_v2.0.00

Published with MATLAB® R2014b

MATLODE_Example_ERK_ADJ_Integrator

Up: Examples



Contents

For the following examples we will use Van Der Pol as a toy problem to illustrate `MATLODE_ERK_ADJ_Integrator` functionalities and features. To initially setup Brusselator, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @vanDerPol_Function;
Ode_Jacobian        = @vanDerPol_Jacobian;
Ode_Lambda          = eye(2);
Ode_Quadrature      = @vanDerPol_Quadrature;
Ode_QFun            = @vanDerPol_QFun;
Ode_DRDP            = @vanDerPol_DRDP;
Ode_DRDY            = @vanDerPol_DRDY;
Ode_Jacp            = @vanDerPol_Jacp;
Ode_Mu              = @vanDerPol_Mu;
Time_Interval       = [ 0 20 ];
Y0                  = [2; -0.66];
```

Basic Functionality

Now that we have our model loaded in our workspace, we can perform an adjoint explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Lambda are required and passed by MATLODE®'s option struct.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda);
[ ~, Y, Sens ] = MATLODE_ERK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
    1.9159   -0.0715
```

```
sensitivity at Time_Interval(2)
    1.4076    0.1210
    0.0470    0.0040
```

Advanced Features

Calculating Mu and Quadrature depends on the input parameters to the option struct. Below are three examples illustrating the required input parameters to obtain the desired output.

Example 1: Mu: false | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Quadrature',Ode_Quadrat
ure,'QFun',Ode_QFun,'DRDY',Ode_DRDY);
[ ~, Y, Sens, Quad, ~ ] = MATLODE_ERK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
```

```
solution at Time_Interval(2)
    1.9159   -0.0715

sensitivity at Time_Interval(2)
   -0.2710    0.5286
   -0.0102    0.0510

quadrature at Time_Interval(2)
    3.5820
   -0.1468
```

Example 2: Mu: true | Quadrature: false

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Jacp',Ode_Jacp,'Mu',Ode
_Mu);
[ ~, Y, Sens, ~, Mu ] = MATLODE_ERK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
    1.9159   -0.0715

sensitivity at Time_Interval(2)
    1.4076    0.1210
    0.0470    0.0040

mu at Time_Interval(2)
    0.1475    0.0199
```

Example 3: Mu: true | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Jacp',Ode_Jacp,'Mu',Ode
_Mu,'Quadrature',Ode_Quadrature,'QFun',Ode_QFun,'DRDY',Ode_DRDY,'DRDP',Ode_DRDP);
[ ~, Y, Sens, Quad, Mu ] = MATLODE_ERK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
    1.9159   -0.0715

sensitivity at Time_Interval(2)
   -0.2710    0.5286
   -0.0102    0.0510

quadrature at Time_Interval(2)
    3.5820
   -0.1468
```

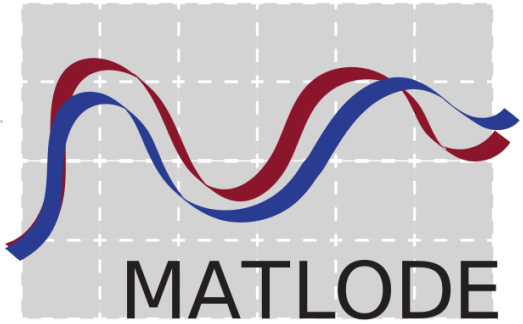
```
  mu at Time_Interval(2)
     -2.9060    0.1674
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_RK_ADJ_Integrator

Up: Examples



Contents

For the following examples we will use Van Der Pol as a toy problem to illustrate `MATLODE_RK_ADJ_Integrator` functionalities and features. To initially setup Brusselator, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @vanDerPol_Function;
Ode_Jacobian        = @vanDerPol_Jacobian;
Ode_Lambda          = eye(2);
Ode_Quadrature      = @vanDerPol_Quadrature;
Ode_QFun            = @vanDerPol_QFun;
Ode_DRDP            = @vanDerPol_DRDP;
Ode_DRDY            = @vanDerPol_DRDY;
Ode_Jacp            = @vanDerPol_Jacp;
Ode_Mu              = @vanDerPol_Mu;
Time_Interval       = [ 0 20 ];
Y0                  = [2; -0.66];
```

Basic Functionality

Now that we have our model loaded in our workspace, we can perform an adjoint implicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Lambda are required and passed by MATLODE®'s option struct.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda);
[ ~, Y, Sens ] = MATLODE_RK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
   1.915826617258943  -0.071569811610919
```

```
sensitivity at Time_Interval(2)
   1.092402436160811   0.070725314893861
   0.036454662784655   0.002360181028024
```

Advanced Features

Calculating Mu and Quadrature depends on the input parameters to the option struct. Below are three examples illustrating the required input parameters to obtain the desired output.

Example 1: Mu: false | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Quadrature',Ode_Quadrat
ure,'QFun',Ode_QFun,'DRDY',Ode_DRDY);
[ ~, Y, Sens, Quad, ~ ] = MATLODE_RK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
```

```
solution at Time_Interval(2)
   1.915826617258943  -0.071569811610919

sensitivity at Time_Interval(2)
   1.844434439987108   0.163127751054598
   0.060418548859476   0.038814843812677

quadrature at Time_Interval(2)
   1.797057140288327
  -0.084173382741058
```

Example 2: Mu: true | Quadrature: false

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Jacp',Ode_Jacp,'Mu',Ode
_Mu);
[ ~, Y, Sens, ~, Mu ] = MATLODE_RK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
   1.915826617258943  -0.071569811610919

sensitivity at Time_Interval(2)
   1.092402436160811   0.070725314893861
   0.036454662784655   0.002360181028024

mu at Time_Interval(2)
   0.118665403858864   0.014805677093766
```

Example 3: Mu: true | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Jacp',Ode_Jacp,'Mu',Ode
_Mu,'Quadrature',Ode_Quadrature,'QFun',Ode_QFun,'DRDY',Ode_DRDY,'DRDP',Ode_DRDP);
[ ~, Y, Sens, Quad, Mu ] = MATLODE_RK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
   1.915826617258943  -0.071569811610919

sensitivity at Time_Interval(2)
   1.844434439987108   0.163127751054598
   0.060418548859476   0.038814843812677

quadrature at Time_Interval(2)
   1.797057140288327
  -0.084173382741058
```

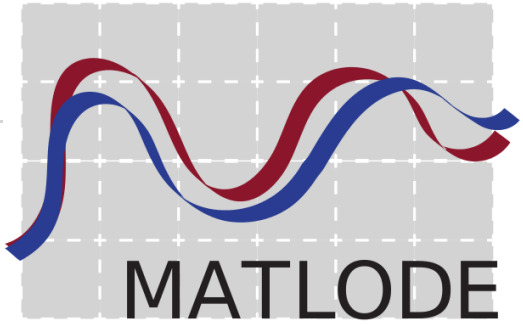
```
mu at Time_Interval(2)
  -2.820453445640490   0.133471080952626
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_ROS_ADJ_Integrator

Up: Examples



Contents

For the following examples we will use Van Der Pol as a toy problem to illustrate `MATLODE_ROS_ADJ_Integrator` functionalities and features. To initially setup Van Der Pol, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @vanDerPol_Function;
Ode_Jacobian        = @vanDerPol_Jacobian;
Ode_Lambda          = eye(2);
Ode_Quadrature      = @vanDerPol_Quadrature;
Ode_QFun            = @vanDerPol_QFun;
Ode_DRDP            = @vanDerPol_DRDP;
Ode_DRDY            = @vanDerPol_DRDY;
Ode_Hesstr_vec      = @vanDerPol_Hesstr_vec;
Ode_Jacp            = @vanDerPol_Jacp;
Ode_Hesstr_vec_r_py = @vanDerPol_Hesstr_vec_r_py;
Ode_Hesstr_vec_f_py = @vanDerPol_Hesstr_vec_f_py;
Ode_Hesstr_vec_r    = @vanDerPol_Hesstr_vec_r;
Ode_Mu              = @vanDerPol_Mu;
Time_Interval       = [ 0 20 ];
Y0                  = [2; -0.66];
```

Basic Functionality

Now that we have our model loaded in our workspace, we can perform an adjoint Rosenbrock integration using MATLODE's prebuilt default settings. We note that a Jacobian and Lambda are required and passed by MATLODE®'s option struct.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Hesstr_vec',Ode_Hesstr
_vec);
[ ~, Y, Sens ] = MATLODE_ROS_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
   1.914997535334449  -0.071620986894423

sensitivity at Time_Interval(2)
   1.153958473571282   0.074770893962760
   0.038510765579949   0.002495310217439
```

Advanced Features

Calculating Mu and Quadrature depends on the input parameters to the option struct. Below are three examples illustrating the required input parameters to obtain the desired output.

Example 1: Mu: false | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda, ...
    'Quadrature',Ode_Quadrature,'QFun',Ode_QFun,'DRDY',Ode_DRDY,...
    'Hesstr_vec',Ode_Hesstr_vec,'Hesstr_vec_r',Ode_Hesstr_vec_r);
[ ~, Y, Sens, Quad, ~ ] = MATLODE_ROS_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
```

```
solution at Time_Interval(2)
   1.914997535334449  -0.071620986894423

sensitivity at Time_Interval(2)
   0.270365153007235   0.228729367534286
   0.007890520586555   0.041006075797396

quadrature at Time_Interval(2)
   1.819228370407761
  -0.085002464665557
```

Example 2: Mu: true | Quadrature: false

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda, ...
    'Jacp',Ode_Jacp,'Mu',Ode_Mu,'NP',1,'Hesstr_vec',Ode_Hesstr_vec,...
```

```
    'Hesstr_vec_f_py',Ode_Hesstr_vec_f_py);
[ ~, Y, Sens, ~, Mu ] = MATLODE_ROS_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
   1.914997535334449  -0.071620986894423

sensitivity at Time_Interval(2)
   1.146931661926977   0.074315588315960
   0.038276263311960   0.002480115529975

mu at Time_Interval(2)
   0.122821616433231   0.015085813689266
```

Example 3: Mu: true | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda, ...
    'Jacp',Ode_Jacp,'Mu',Ode_Mu,'Quadrature',Ode_Quadrature, ...
    'QFun',Ode_QFun,'DRDY',Ode_DRDY,'DRDP',Ode_DRDP,'NP',1, ...
    'Hesstr_vec',Ode_Hesstr_vec,'Hesstr_vec_f_py',Ode_Hesstr_vec_f_py, ...
    'Hesstr_vec_r_py',Ode_Hesstr_vec_r_py,'Hesstr_vec_r',Ode_Hesstr_vec_r);
[ ~, Y, Sens, Quad, Mu ] = MATLODE_ROS_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
   1.914997535334449  -0.071620986894423
```

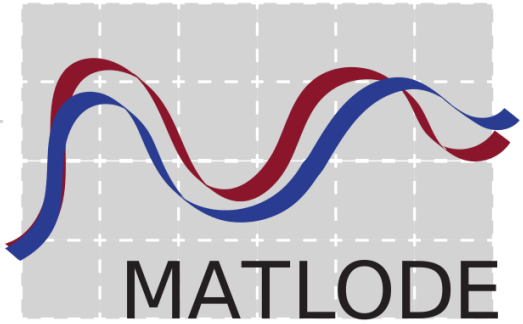
```
sensitivity at Time_Interval(2)
   0.447846833780721   0.221247250242815
   0.013813572620501   0.040756378841930

quadrature at Time_Interval(2)
   1.819228370407761
  -0.085002464665557

mu at Time_Interval(2)
  -5.831338011237693   0.276533426870627
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_SDIRK_ADJ_Integrator

Up: Examples



Contents

For the following examples we will use Van Der Pol as a toy problem to illustrate `MATLODE_SDIRK_ADJ_Integrator` functionalities and features. To initially setup Brusselator, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @vanDerPol_Function;
Ode_Jacobian        = @vanDerPol_Jacobian;
Ode_Lambda          = eye(2);
Ode_Quadrature      = @vanDerPol_Quadrature;
Ode_QFun            = @vanDerPol_QFun;
Ode_DRDP            = @vanDerPol_DRDP;
Ode_DRDY            = @vanDerPol_DRDY;
Ode_Jacp            = @vanDerPol_Jacp;
Ode_Mu              = @vanDerPol_Mu;
Time_Interval       = [ 0 20 ];
Y0                  = [2; -0.66];
```

Basic Functionality

Now that we have our model loaded in our workspace, we can perform an adjoint singly diagonally implicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Lambda are required and passed by MATLODE®'s option struct.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda);
[ ~, Y, Sens ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
   1.915916039794937  -0.071568494586575
```

```
sensitivity at Time_Interval(2)
   1.074077523765376   0.069576088307242
   0.035843346868728   0.002321843453369
```

Advanced Features

Calculating Mu and Quadrature depends on the input parameters to the option struct. Below are three examples illustrating the required input parameters to obtain the desired output.

Example 1: Mu: false | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Quadrature',Ode_Quadrat
ure,'QFun',Ode_QFun,'DRDY',Ode_DRDY);
[ ~, Y, Sens, Quad, ~ ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
```

```
solution at Time_Interval(2)
   1.915916039794937  -0.071568494586575

sensitivity at Time_Interval(2)
   2.283745365862151   0.143653612072566
   0.075079253248356   0.038165190322096

quadrature at Time_Interval(2)
   1.794841899941942
  -0.084083960205060
```

Example 2: Mu: true | Quadrature: false

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Jacp',Ode_Jacp,'Mu',Ode
_Mu,'NP',1);
[ ~, Y, Sens, ~, Mu ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
   1.915916039794937  -0.071568494586575

sensitivity at Time_Interval(2)
   1.074077523765443   0.069576088307223
   0.035843346868730   0.002321843453369

mu at Time_Interval(2)
   0.117269908710761   0.014719679001328
```

Example 3: Mu: true | Quadrature: true

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Lambda',Ode_Lambda,'Jacp',Ode_Jacp,'Mu',Ode
_Mu,'Quadrature',Ode_Quadrature,'QFun',Ode_QFun,'DRDY',Ode_DRDY,'DRDP',Ode_DRDP,'NP',1);
[ ~, Y, Sens, Quad, Mu ] = MATLODE_SDIRK_ADJ_Integrator(Ode_Function,Time_Interval,Y0,Options)
;
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
disp('quadrature at Time_Interval(2)');
disp(Quad);
disp('mu at Time_Interval(2)');
disp(Mu);
```

```
solution at Time_Interval(2)
   1.915916039794937  -0.071568494586575

sensitivity at Time_Interval(2)
   2.283745365890121   0.143653612072536
   0.075079253249290   0.038165190322095

quadrature at Time_Interval(2)
   1.794841899941942
  -0.084083960205060
```

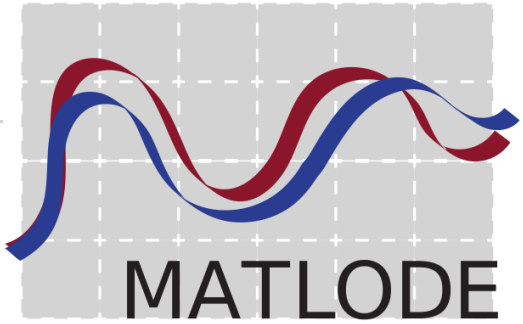
```
  mu at Time_Interval(2)
    -2.795903994247277   0.131989587712079
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_ERK_FWD_Integrator

Up: Examples



Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_ERK_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward explicit Runge-Kutta integration using the prebuilt default settings.

```
[ ~, Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function,Time_Interval,Y0);
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
```

```
solution at Time_Interval(2)
    0.9894   -0.0081   -1.1139   -1.3474
```

Advanced Features

To **save the model state at each time step**, one needs to initialize a MATLODE® option struct to store the fine tuning settings. The `(key,value)` pair associated for saving the model state at each time step is denoted as `('storeCheckpoint',true)` or `('storeCheckpoint',false)` depending on whether or not one wants to explicitly fine tune the integrator. In this case, the intermediary time step values are stored executing the command below.

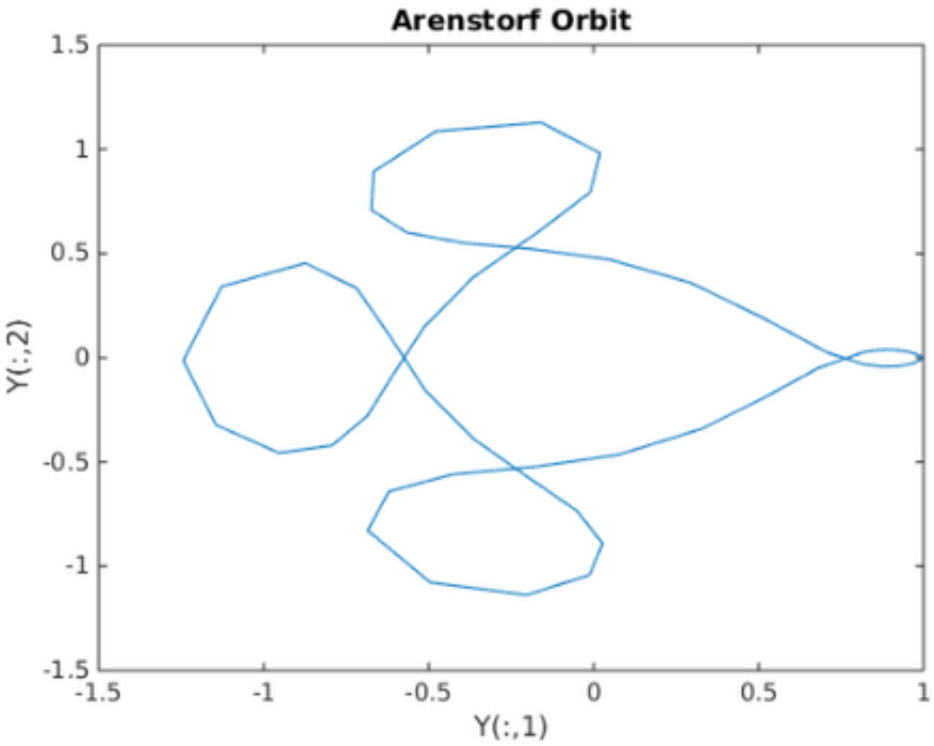
```
Options = MATLODE_OPTIONS('storeCheckpoint',true);
```

To run `MATLODE_ERK_FWD_Integrator` using the fine tuning, one needs to insert the option struct into the integrator's fourth parameter position.

```
[ ~, Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

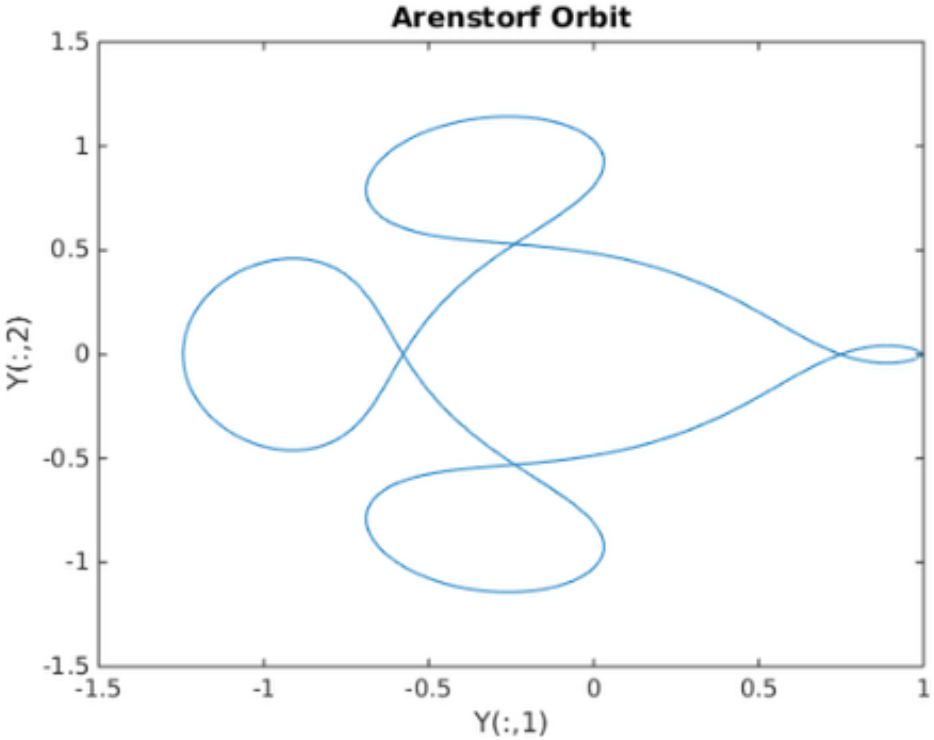
After plotting the results, one can now visualize the model.

```
figure(1);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
```



To obtain a smoother graphical repusentation, one can further tighten the error tolerances. To tighten the relative and abolute error tolerances, one fine tunes the option struct. Since the option struct is already in the workspace, one adds the relative and absolute `(key,value)` pair to the option struct. Then plot the results.

```
Options = MATLODE_OPTIONS(Options,'AbsTol',1e-12,'RelTol',1e-12);
[ T, Y ] = MATLODE_ERK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
figure(2);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
```



Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_RK_FWD_Integrator

Up: Examples

Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_RK_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward implicit Runge-Kutta integration using the prebuilt default settings. Note, for implicite Runge-Kutta integrators, the Jacobian is required.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian);
[ ~, Y ] = MATLODE_RK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.995683236985069   0.003082830712656   0.337097245941238

  Column 4

  -1.690210404461741
```

Advanced Features

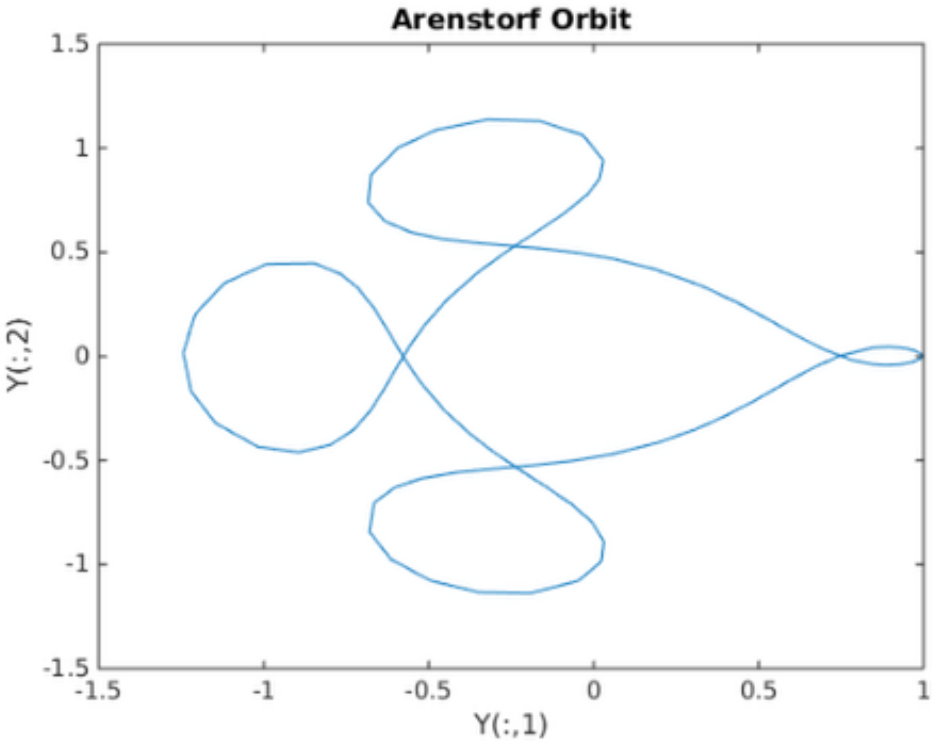
To **save the model state at each time step**, one needs to initialize a MATLODE® option struct to store the fine tuning settings. The `(key,value)` pair associated for saving the model state at each time step is denoted as `('storeCheckpoint',true)` or `('storeCheckpoint',false)` depending on whether or not one wants to explicitly fine tune the integrator. In this case, the intermediary time step values are stored executing the command below.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'storeCheckpoint',true);
[ ~, Y, Stats ] = MATLODE_RK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

After plotting the results, one can now visualize the model.

```
figure(1);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
PrintISTATUS(Stats.ISTATUS);
```

```
ISTATUS =

    Nfun:    396
    Njac:    113
    Nstp:    130
    Nacc:    113
    Nrej:     12
    Ndec:    130
    Nsol:    857
    Nsng:      0
```



Depending on the model, it may be advantageous to use Kjell Gustafsson's error control approach described in [1]. One notes that in this toy problem, it is better to use Gusafsson, but for illustration purposes Gustafsson is toggled off to demonstrate the effect.

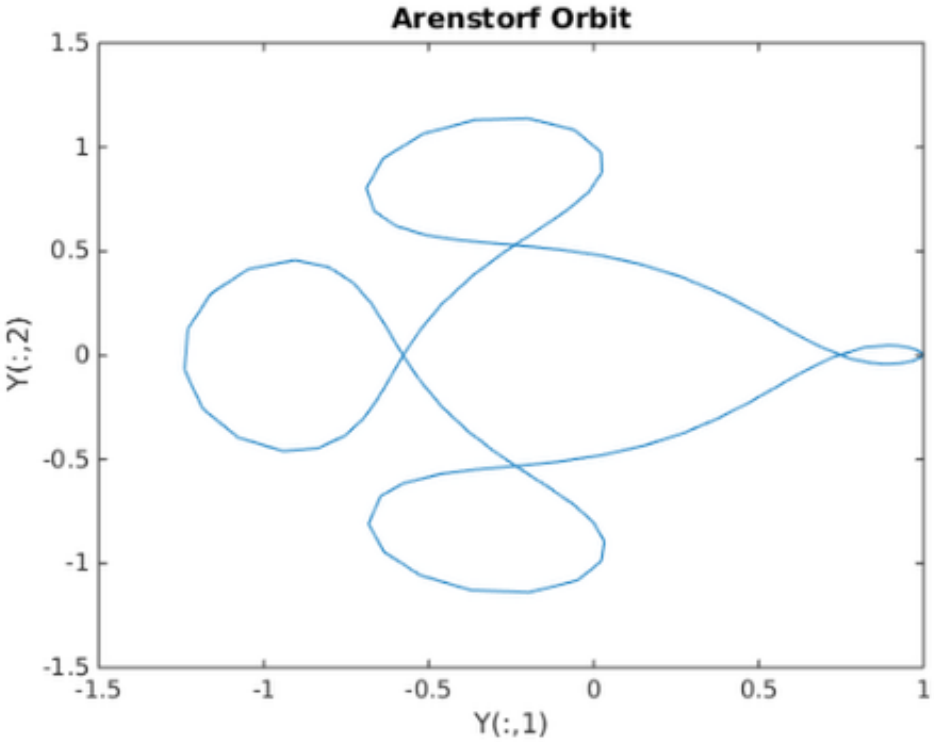
```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'storeCheckpoint',true,'Gustafsson',false);
[ ~, Y, Stats ] = MATLODE_RK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

After plotting the results and printing the integrator statistics, one can visualize the model and compare statistics to the previous example.

```
figure(2);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
PrintISTATUS(Stats.ISTATUS);
```

```
ISTATUS =

    Nfun:    429
    Njac:    110
    Nstp:    143
    Nacc:    110
    Nrej:     25
```

```
Ndec:    143
Nsol:    951
Nsng:      0
```



Reference

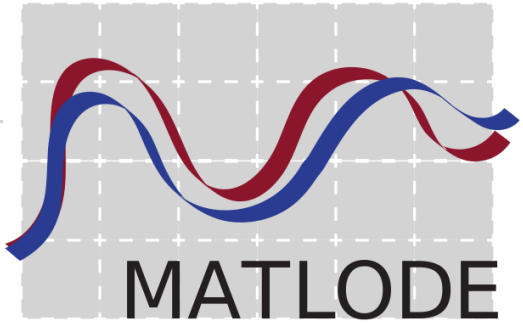
[1] K. Gustafsson, Control of error and convergence in ODE solvers, 1992 :Dept. of Automat. Contr., Lund Inst. Technol.

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_ROS_FWD_Integrator

Up: Examples



Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_ROS_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward Rosenbrock integration using the prebuilt default settings. Note, for Rosenbrock integrators, the Jacobian is required.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian);
[ ~, Y ] = MATLODE_ROS_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.991128738194503   0.005808680341908   0.933220592793551

  Column 4

  -1.694307055801689
```

Advanced Features

To analyze intermediary steps, toggle the option struct parameter 'displaySteps' to true. Printing intermediary steps often gives an immediate visual queue on how hard the intergrator is working internally.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'displaySteps',true);
[ ~, Y ] = MATLODE_ROS_FWD_Integrator(Ode_Function,[Time_Interval(1) Time_Interval(1)+0.0029],
Y0,Options);
```

```
Accepted step. Time = 1e-05; Stepsize = 6e-05
Accepted step. Time = 7e-05; Stepsize = 0.00036
Accepted step. Time = 0.00043; Stepsize = 0.00077939
Accepted step. Time = 0.0012094; Stepsize = 0.00073114
Accepted step. Time = 0.0019405; Stepsize = 0.0011091
Accepted step. Time = 0.0029; Stepsize = 0.00093643
```

Noticing above that no initial steps are rejected, increasing Hstart will force the error controller to be more aggressive in the beginning of the Rosenbrock intergration scheme.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'displaySteps',true,'Hstart',0.0005);
[ ~, Y ] = MATLODE_ROS_FWD_Integrator(Ode_Function,[Time_Interval(1) Time_Interval(1)+0.0029],
Y0,Options);
```

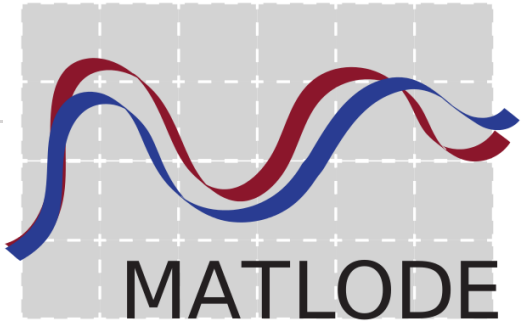
```
Accepted step. Time = 0.0005; Stepsize = 0.00069527
Accepted step. Time = 0.0011953; Stepsize = 0.00079987
Accepted step. Time = 0.0019951; Stepsize = 0.0010336
Accepted step. Time = 0.0029; Stepsize = 0.00093982
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_SDIRK_FWD_Integrator

Up: Examples



Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_SDIRK_FWD_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_JacobianVector  = @arenstorfOrbit_JacobianVector;
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward Rosenbrock integration using the prebuilt default settings.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian);
[ ~, Y ] = MATLODE_SDIRK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.993361350358526  -0.003791231129465  -0.557175919928586

  Column 4

  -1.843226156724899
```

Advanced Functionality

If an analytical Jacobian is not available, MATLODE® can approximate the Jacobian by toggling the 'MatrixFree' option parameter.

```
Options = MATLODE_OPTIONS('MatrixFree',true);
[ ~, Y ] = MATLODE_SDIRK_FWD_Integrator(Ode_Function,[Time_Interval(1) Time_Interval(1)+0.1],Y
0,Options);
```

```
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.000111;     H=0.0003073
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.00026465;     H=0.00029234
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.00041082;     H=0.00029336
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.0005575;     H=0.00029889
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.00070694;     H=0.00030641
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.00086015;     H=0.00031547
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.0010418;     H=0.0003253
Warning: Input tol may not be achievable by GMRES.
 Try to use a bigger tolerance.
Warning: GMRES: stagnated (two consecutive iterates were the same)
MATRIX IS SINGULAR , ISING=1;     T=0.0012291;     H=0.00033967
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
```

```
solution at Time_Interval(2)
```

```
    Columns 1 through 3

     0.918124438055006  -0.039973777648758  -0.638924935480216

    Column 4

    -0.088581558035649
```

If a Jacobian vector product approximation is available, can pass the Jacobian vector product function handler to 'Jacobian' and toggle the 'MatrixFree' in the option struct.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_JacobianVector,'MatrixFree',true);
[ ~, Y ] = MATLODE_SDIRK_FWD_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Execute the following commands to analyze the final model state.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
```

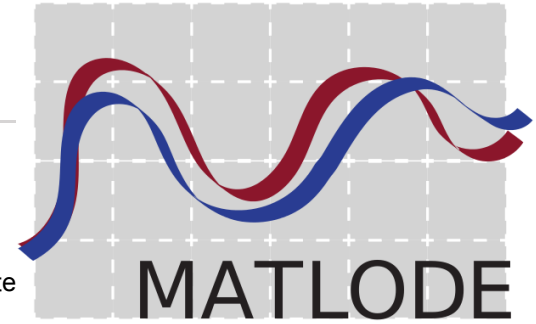
```
  solution at Time_Interval(2)
    Columns 1 through 3

     0.996883951675026   0.020584159426189   0.633482175864490

    Column 4

    -0.884556217536157
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_ERK_TLM_Integrator

Up: Examples



Contents

For the following examples we will use Arenstorf Orbit as a toy problem to illustrate `MATLODE_ERK_TLM_Integrator` functionalities and features. To initially setup Brusselator, execute the MATLAB commands below to load our input parameters into our workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that we have our model loaded in our workspace, we can perform a tangent linear explicit Runge-Kutta integration using MATLODE's prebuilt default settings. We note that a Jacobian and Y_TLM required and passed by MATLODE's option struct.

```
Options  = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM);
[ ~, Y, Sens ] = MATLODE_ERK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
    0.9894   -0.0081   -1.1139   -1.3474

sensitivity at Time_Interval(2)
   1.0e+06 *

    0.0139   -0.0043    0.0000   -0.0001
    0.0082   -0.0023    0.0000   -0.0001
    0.9444   -0.2974    0.0019   -0.0059
```

```
    -1.4832    0.4407   -0.0028    0.0092
```

Advanced Features

To **save the model state at each time step**, one needs to initialize a MATLODE® option struct to store the fine tuning settings. The `(key,value)` pair associated for saving the model state at each time step is denoted as `('storeCheckpoint',true)` or `('storeCheckpoint',false)` depending on whether or not one wants to explicitly fine tune the integrator. In this case, the intermediary time step values are stored executing the command below.

```
Options = MATLODE_OPTIONS('storeCheckpoint',true,'Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM);
```

To run `MATLODE_ERK_FWD_Integrator` using the fine tuning, one needs to insert the option struct into the integrator's fourth parameter position.

```
[ ~, Y, Sens ] = MATLODE_ERK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

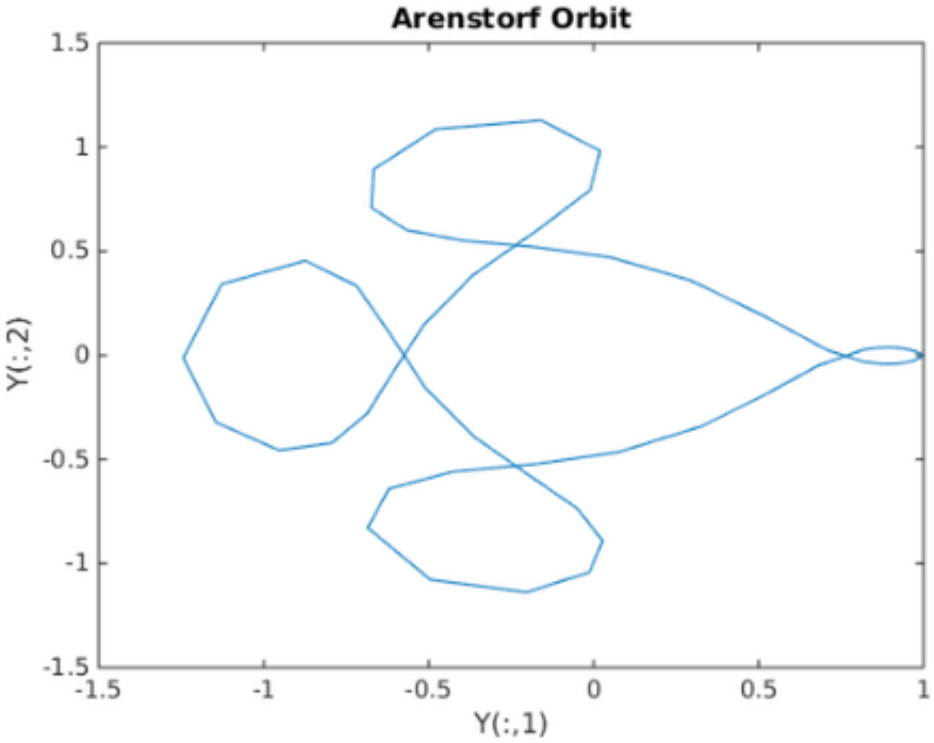
```
format long;
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.989365398322643  -0.008118074065802  -1.113854858951780

  Column 4

  -1.347375530505811

sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3

   0.013935972017972  -0.004266173814348   0.000027043915070
   0.008179760777190  -0.002348774863634   0.000014891733314
   0.944413955565086  -0.297449809968606   0.001885443761994
  -1.483218412779525   0.440717675442452  -0.002793988009634

  Column 4
```

```
   -0.000086652911918
   -0.000050959021233
   -0.005867049814353
    0.009230958322628
```

After plotting the results, one can now visualize the model.

```
figure(1);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
```



To obtain a smoother graphical repusentation, one can further tighten the error tolerances. To tighten the relative and abolute error tolerances, one fine tunes the option struct. Since the option struct is already in the workspace, one adds the relative and absolute `(key,value)` pair to the option struct. Then plot the results.

```
Options = MATLODE_OPTIONS(Options,'AbsTol',1e-12,'RelTol',1e-12);
[ T, Y, Sens ] = MATLODE_ERK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

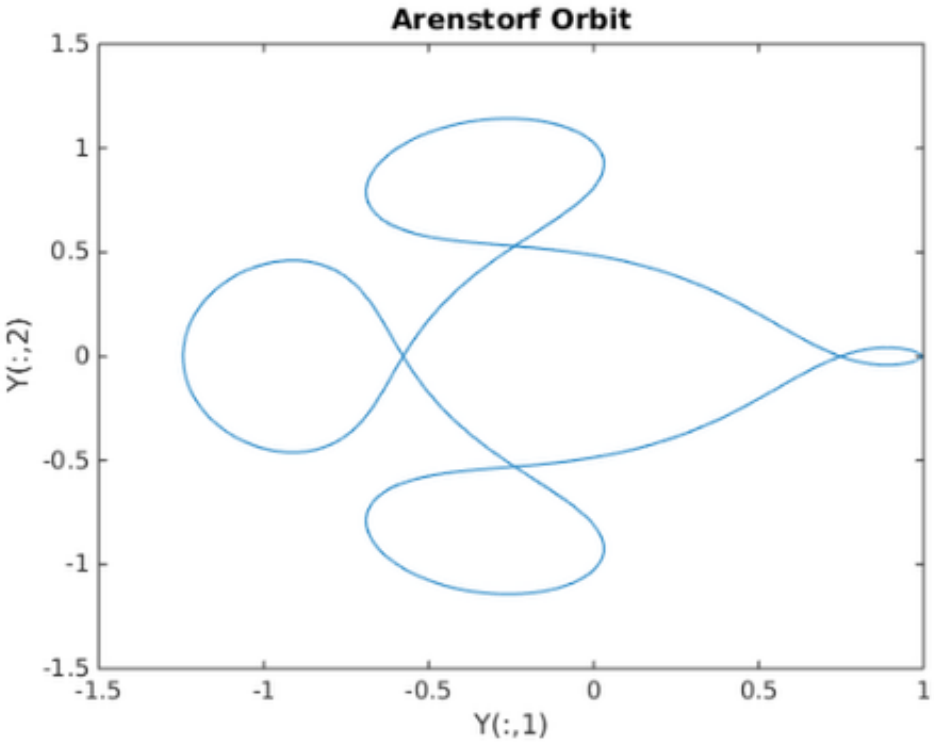
```
format long;
```

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.993999999922114  -0.000000079957168  -0.000012606226297

  Column 4

  -2.001585118385729

sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3

   0.004141460842001  -0.001373436793264   0.000008712126201
   0.013650742272501  -0.004049058454606   0.000025690749214
   2.220300498658203  -0.660821033958704   0.004192788254230
   0.644407485994680  -0.213758538185508   0.001355935230860

  Column 4

  -0.000025691247242
  -0.000084980825140
  -0.013820780308166
  -0.003997499017851
```

After plotting the results, one can now visualize the model.

```
figure(2);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
```

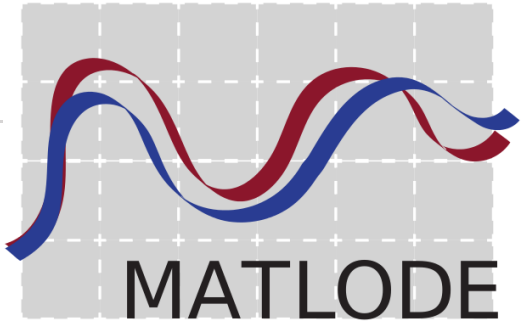


Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_RK_TLM_Integrator

Up: Examples



Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_RK_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward implicit Runge-Kutta integration using the prebuilt default settings. Note, for implicite Runge-Kutta integrators, the Jacobian is required.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM);
[ ~, Y, Sens ] = MATLODE_RK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.995683236985069   0.003082830712656   0.337097245941238

  Column 4

  -1.690210404461741
```

```
sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3

   0.001897034914603  -0.000702632999756   0.000004457258758
   0.014014939823589  -0.004225978530536   0.000026821010152
   1.129083784775020  -0.334712638502328   0.002124411988722
   0.904569072954531  -0.283792487614403   0.001800964778757

  Column 4

  -0.000011725290480
  -0.000087222290942
  -0.007030445866727
  -0.005622766156361
```

Advanced Features

To perform **nondirect sensitivity analysis**, toggle the 'DirectTLM' option parameter to false. This enables the sensitivity matrix to be calculated using Newton iterations. Note, it is strongly recommended to first try direct sensitivity analysis before trying nondirect for efficiency purposes.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM,'DirectTLM',false);
[ ~, Y, Sens ] = MATLODE_RK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.995683236985069   0.003082830712656   0.337097245941238

  Column 4

  -1.690210404461741

sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3
```

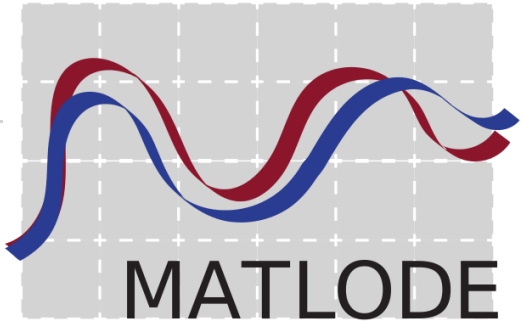
```
    0.001897073110606  -0.000702641110081   0.000004457308369
    0.014015303195956  -0.004226093249392   0.000026821726256
    1.129113450285306  -0.334722165479576   0.002124471499005
    0.904591601665614  -0.283799265344582   0.001801007001283

  Column 4

  -0.000011725535564
  -0.000087224584148
  -0.007030632922057
  -0.005622908673807
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_ROS_TLM_Integrator

Up: Examples



Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_ROS_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_HessVec         = @arenstorfOrbit_Hess_vec;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward implicit Runge-Kutta integration using the prebuilt default settings. Note, for Rosenbrock integrators, the Jacobian, Y_TLM and HessVec is required.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM,'Hess_vec',Ode_HessVec);
[ ~, Y, Sens ] = MATLODE_ROS_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.993966585991924  -0.000098537235715  -0.016200670776286

  Column 4

  -2.006618134552174
```

```
sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3

   0.004246508785152  -0.001404493356811   0.000008909134150
   0.013649774590984  -0.004047724277430   0.000025682157554
   2.252748424954262  -0.670840906667980   0.004256333758228
   0.613990351650225  -0.204877928488845   0.001299580559965

  Column 4

  -0.000026345289476
  -0.000084975507507
  -0.014022541136757
  -0.003808054887770
```

Advanced Features

To perform **nondirect sensitivity analysis**, toggle the 'DirectTLM' option parameter to false. This enables the sensitivity matrix to be calculated using Newton iterations. Note, it is strongly recommended to first try direct sensitivity analysis before trying nondirect for efficiency purposes.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM,'Hess_vec',Ode_HessVec,'Dir
ectTLM',false);
[ ~, Y, Sens ] = MATLODE_ROS_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.993966585991924  -0.000098537235715  -0.016200670776286

  Column 4

  -2.006618134552174

sensitivity at Time_Interval(2)
   1.0e+06 *
```

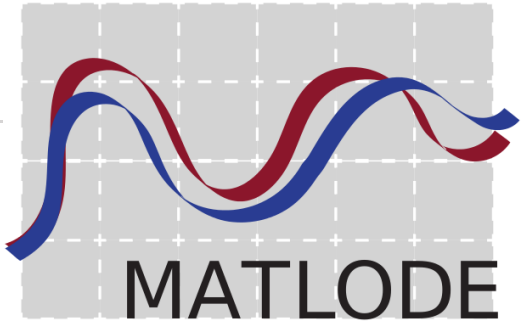
```
Columns 1 through 3

 0.004246508785152  -0.001404493356811   0.000008909134150
 0.013649774590984  -0.004047724277430   0.000025682157554
 2.252748424954262  -0.670840906667980   0.004256333758228
 0.613990351650225  -0.204877928488845   0.001299580559965

Column 4

-0.000026345289476
-0.000084975507507
-0.014022541136757
-0.003808054887770
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b

MATLODE_Example_SDIRK_TLM_Integrator

Up: Examples



Contents

For the following examples Arenstorf Orbit is used as a toy problem to illustrate `MATLODE_SDIRK_TLM_Integrator` functionalities and features. To initially setup Arenstorf Orbit, execute the MATLAB commands below to load the input parameters into the workspace.

```
Ode_Function        = @arenstorfOrbit_Function;
Ode_Jacobian        = @arenstorfOrbit_Jacobian;
Ode_YTLM            = eye(4);
Time_Interval       = [ 0 17.0652166 ];
Y0                  = [0.994; 0; 0; -2.00158510637908252240537862224];
```

Basic Functionality

Now that the model is loaded in the workspace, one performs a forward implicit Runge-Kutta integration using the prebuilt default settings. Note, for implicite Runge-Kutta tangent linear integrators, the Jacobian and Y_TLM is required.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM,'storeCheckpoint',true);
[ ~, Y, Sens ] = MATLODE_SDIRK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

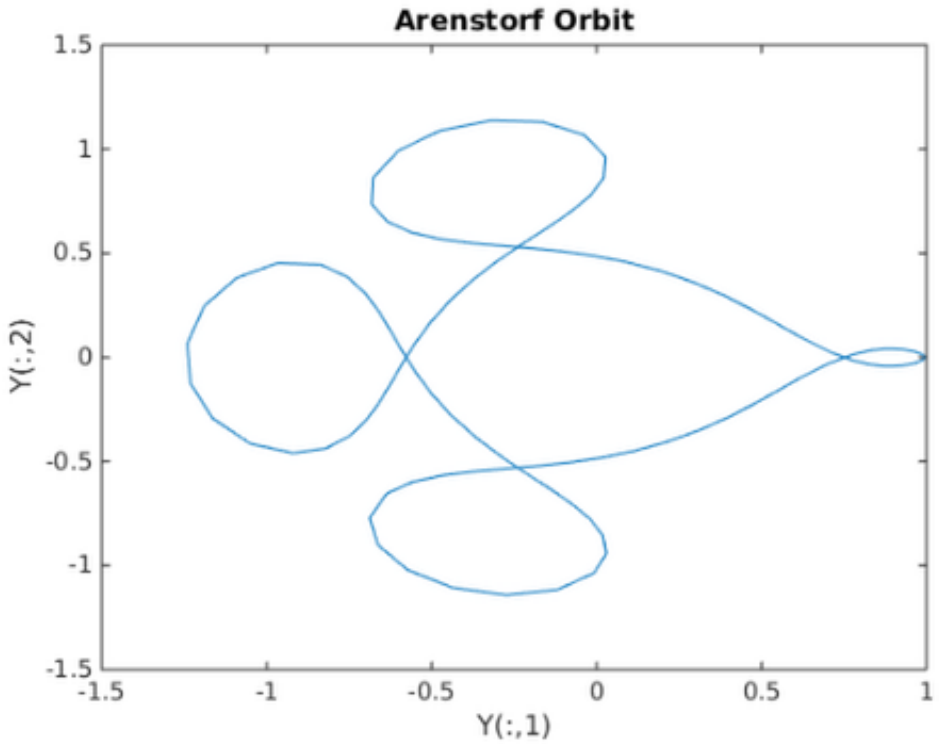
```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.993361350358526  -0.003791231129465  -0.557175919928586

  Column 4

  -1.843226156724899
```

```
sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3

   0.008664706038234  -0.002746472000195   0.000017424543948
   0.013389672332976  -0.004012360248571   0.000025459057471
   2.192100809653940  -0.675013681548083   0.004282793806521
  -0.703509778005246   0.198286030016517  -0.001258342928550

  Column 4

  -0.000053831489093
  -0.000083331076401
  -0.013631308567663
   0.004386138901873
```

After plotting the results, one can now visualize the model.

```
figure(1);
plot(Y(:,1),Y(:,2));
title('Arenstorf Orbit');
xlabel('Y(:,1)');
ylabel('Y(:,2)');
```



Advanced Features

To perform **nondirect sensitivity analysis**, toggle the 'DirectTLM' option parameter to false. This enables the sensitivity matrix to be calculated using Newton iterations. Note, it is strongly recommended to first try direct sensitivity analysis before trying nondirect for efficiency purposes.

```
Options = MATLODE_OPTIONS('Jacobian',Ode_Jacobian,'Y_TLM',Ode_YTLM,'DirectTLM',false);
[ ~, Y, Sens ] = MATLODE_SDIRK_TLM_Integrator(Ode_Function,Time_Interval,Y0,Options);
```

Printing out our results, we can analyze our model state at our final time.

```
disp('solution at Time_Interval(2)');
disp(Y(end,:));
disp('sensitivity at Time_Interval(2)');
disp(Sens);
```

```
solution at Time_Interval(2)
  Columns 1 through 3

   0.993361350358526  -0.003791231129465  -0.557175919928586

  Column 4

  -1.843226156724899

sensitivity at Time_Interval(2)
   1.0e+06 *

  Columns 1 through 3

   0.008664653756679  -0.002746451121725   0.000017424413239
   0.013389579093279  -0.004012322626933   0.000025458821407
   2.192086619721465  -0.675007972703288   0.004282758023376
  -0.703504330964360   0.198283802514363  -0.001258328929495

  Column 4

  -0.000053831163761
  -0.000083330497371
  -0.013631220367666
   0.004386105130705
```

Authored by Tony D'Augustine, Adrian Sandu, and Hong Zhang.
Computational Science Laboratory, Virginia Tech.

Published with MATLAB® R2014b